CHEMNITZ UNIVERSITY OF TECHNOLOGY

Faculty of Computer Science

Computer Architecture Group

Diploma Thesis

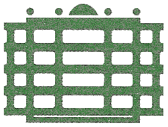
Concepts and Prototype for a Collective Offload Unit –GOAL on EXTOLL–

Timo Schneider, Sven Eckelmann

Chemnitz, December 1, 2011

Superadvisor: Prof. Dr.-Ing. Wolfgang Rehm
Advisor: Prof. Dr. rer. nat. Torsten Hoefler
Dipl.-Inf. Jochen Strunk

Timo Schneider, Sven Eckelmann
Concepts and Prototype for a Collective Offload Unit
Diploma Thesis, Faculty of Computer Science
Chemnitz University of Technology, 2011



TECHNISCHE UNIVERSITÄT
CHEMNITZ

Aufgabenstellung

zur

Diplomarbeit
im Studiengang Informatik

für

Herrn Sven Eckelmann
geb. am 31. Juli 1984 in Karl-Marx-Stadt

zum Thema

Concepts and Prototype for a Collective Offload Unit -GOAL on EXTOLL-

(ausführliche Aufgabenstellung siehe Rückseite)

Erstgutachter:	Prof. Dr. Wolfgang Rehm
Ausgabedatum:	01.09.2011
Abgabedatum:	29.02.2012
Tag der Abgabe:	

Unterschrift: .....

Prof. Dr. F. Hamker
Vorsitzender des Prüfungsausschusses

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Aufgabenstellung

zur

Diplomarbeit
im Studiengang Informatik

für

Herrn Timo Schneider
geb. am 19. Dezember 1984 in Göppingen

zum Thema

Concepts and Prototype for a Collective Offload Unit -GOAL on EXTOLL-

(ausführliche Aufgabenstellung siehe Rückseite)

Erstgutachter:	Prof. Dr. Wolfgang Rehm
Ausgabedatum:	01.09.2011
Abgabedatum:	29.02.2012
Tag der Abgabe:	

Unterschrift: .....

Prof. Dr. F. Hamker
Vorsitzender des Prüfungsausschusses

Ausführliche Aufgabenstellung:

The steady increase of parallelism in high-performance computing (HPC) systems implies that communication is and will be most important for large-scale applications. One of the most important communication operations in today's HPC applications are collective communications. Using non-blocking collectives (NBCs), communication can be overlapped with computation and thus deliver higher application performance. To achieve better performance, the offloading of NBCs has been proposed and recent HPC interconnects offer hardware support for such offload, each using its own proprietary interface. The Group Operation Assembly Language (GOAL) has been proposed to provide an unified interface for the specification of, and an execution model for, NBCs. The EXTOLL network interface cards (NIC) are a suitable platform to evaluate concepts for NBC offload as they are implemented with Field Programmable Gate Arrays (FPGAs) and thus allow to change parts of the design.

Topic of this thesis is the design and the prototypical implementation of a dependency driven framework, as proposed by GOAL, which allows offloading NBC operations to NICs. The collectives are defined by user processes as a set of primitive network operations and the dependencies between them. GOAL represents arbitrary collective communication schemes by translating them into a graph. Vertices represent primitive network operations, such as send and receive as well as arithmetic operations on local data. Edges represent the dependencies between those operations.

Since resources such as memory are more constrained for an offload unit than for a traditional, software based message-passing framework running on the host CPU, it has to be investigated if the execution of large graphs with an offloaded GOAL interpreter is possible despite those limited resources. A functional unit capable of interpreting such a graph has to keep track of the dependencies of each outstanding operation, match incoming data to posted receive operations and handle point to point communications with a large number of peers . It has to be ensured that the collective interpreter can always make progress and resource exhaustion does not lead to deadlocks. This is the main theoretical problem of this thesis. To prove that the concepts proposed in this thesis are applicable in practice a prototype should be implemented using a hardware description language (HDL) and an FPGA.

The main practical problem is the implementation of a system with the described complexity on the low level of abstraction necessary to achieve good performance on an FPGA.

The following aspects should be dealt with:

* Identify important properties of well-known collective communication schemes that could influence the design of the microarchitecture for a collective communication offload unit.

* Investigate how arbitrarily large communication schedules can be transformed to become executable by a collective offload unit with limited buffers.

* Develop concepts for the microarchitecture of a communication schedule interpreter that take the scarcity of resources, such as memory, into account. For example, the point-to-point messaging protocol used by such an on-NIC interpreter unit cannot use buffers to store unexpected messages as traditional eager-protocols do.

* Implement the proposed collective communication offload unit as an FPGA synthesizable model in a HDL. However, interfacing with the EXTOLL network and the EXTOLL NIC is not part of this thesis.

The main tasks of Timo Schneider for this thesis are the analysis of the properties of the chosen architecture for collective offload, as well as the implementation of a matching unit in Verilog. Therefore it is important to analyze existing matching functions implemented on host CPUs. Since the proposed architecture can not leverage large message buffers, as they are available to software based message-passing frameworks, the point-to-point protocol can not rely on such buffers. Therefore a new point-to-point protocol has to be developed. To make sure that protocol is deadlock free it has to be verified.

The main tasks of Sven Eckelmann for this thesis are the analysis of the resource requirements of typical collective communication schedules graph representations as well as the Verilog implementation of the functional unit responsible for the point-to-point protocol and the functional unit responsible for starting new operations. While implementing the point-to-point protocol it is important to continuously refine its verification model. Another important aspect of his work is keeping track of the expected performance of each functional unit by analyzing their timing behavior by simulation.

Contents

Abstract

Optimized implementations of blocking and nonblocking collective operations are most important for scalable high-performance applications. Offloading such collective operations into the communication layer can improve performance and asynchronous progression of the operations. However, it is most important that such offloading schemes remain flexible in order to support user-defined (sparse neighbor) collective communications. In this work we propose a design for a collective offload unit. Our hardware design is able to execute dependency graph based representations of collective functions. To cope with the scarcity of memory resources we designed a new point to point messaging protocol which does not need to store information about unexpected messages. The offload unit proposed in this thesis could be integrated into high performance networks such as EXTOLL. Our design achieves a clock frequency of 212 MHz on a Xilinx Virtex6 FPGA, while using less than 10% of the available logic slices and less than 30% of the available memory blocks. Due to the specialization of our design we can accelerate important tasks of the message passing framework, such as message matching by a factor of two, compared to a software implementation running on a CPU with a ten times higher clock speed.

1. Task Description

The steady increase of parallelism in high-performance computing (HPC) systems implies that communication is and will be most important for large-scale applications. One of the most important communication operations in today's HPC applications are collective communications. Using non-blocking collectives (NBCs), communication can be overlapped with computation and thus deliver higher application performance. To achieve better performance, the offloading of NBCs has been proposed and recent HPC interconnects offer hardware support for such offload, each using its own proprietary interface. The Group Operation Assembly Language (GOAL) has been proposed to provide an unified interface for the specification of, and an execution model for, NBCs. The EXTOLL network interface cards (NIC) are a suitable platform to evaluate concepts for NBC offload as they are implemented with Field Programmable Gate Arrays (FPGAs) and thus allow to change parts of the design.

Topic of this thesis is the design and the prototypical implementation of a dependency driven framework, as proposed by GOAL, which allows offloading NBC operations to NICs. The collectives are defined by user processes as a set of primitive network operations and the dependencies between them. GOAL represents arbitrary collective communication schemes by translating them into a graph. Vertices represent primitive network operations, such as send and receive as well as arithmetic operations on local data. Edges represent the dependencies between those operations.

Since resources such as memory are more constrained for an offload unit than for a traditional, software based message-passing framework running on the host CPU, it has to be investigated if the execution of large graphs with an offloaded GOAL interpreter is possible despite those limited resources. A functional unit capable of interpreting such a graph has to keep track of the dependencies of each outstanding operation, match incoming data to posted receive operations and handle point to point communications with a large number of peers . It has to be ensured that the collective interpreter can always make progress and resource exhaustion does not lead to deadlocks. This is the main theoretical problem of this thesis. To prove that the concepts proposed in this thesis are applicable in practice a prototype should be implemented using a hardware description language (HDL) and an FPGA.

The main practical problem is the implementation of a system with the described complexity on the low level of abstraction necessary to achieve good performance on an FPGA.

The following aspects should be dealt with:

- Identify important properties of well-known collective communication schemes that could influence the design of the microarchitecture for a collective communication offload unit.
- Investigate how arbitrarily large communication schedules can be transformed to become executable by a collective offload unit with limited buffers.
- Develop concepts for the microarchitecture of a communication schedule interpreter that take the scarcity of resources, such as memory, into account. For example, the point-to-point messaging protocol used by such an on-NIC interpreter unit cannot use buffers to store unexpected messages as traditional eager-protocols do.
- Implement the proposed collective communication offload unit as an FPGA synthesizable model in a HDL. However, interfacing with the EXTOLL network and the EXTOLL NIC is not part of this thesis.

The main tasks of Timo Schneider for this thesis are the analysis of the properties of the chosen architecture for collective offload, as well as the implementation of a matching unit in Verilog. Therefore it is important to analyze existing matching functions implemented on host CPUs. Since the proposed architecture cannot leverage large message buffers, as they are available to software based message-passing frameworks, the point-to-point protocol cannot rely on such buffers. Therefore a new point-to-point protocol has to be developed. To make sure that protocol is deadlock free it has to be verified.

The main tasks of Sven Eckelmann for this thesis are the analysis of the resource requirements of typical collective communication schedules graph representations as well as the Verilog implementation of the functional unit responsible for the point-to-point protocol and the functional unit responsible for starting new operations. While implementing the point-to-point protocol it is important to continuously refine its verification model. Another important aspect of his work is keeping track of the expected performance of each functional unit by analyzing their timing behavior by simulation.

1.1. Theses

In this work we will examine the following theses:

- It is possible to interpret a dependency graph based representation of a collective function on an FPGA. Such an interpreter unit can be used to offload the execution of arbitrary communication patterns from the host CPU to a network interface card.

- Since the hardware resources available on an FPGA are different from those available to a host CPU such an interpreter cannot be implemented in the same way as in software.

- A dependency graph based schedule interpreter implemented on a Virtex6 FPGA can achieve clock speeds around 200 MHz, suggesting that ASIC implementations with much higher clock speeds are achievable.

- Such an offload unit can provide a performance advantage over host CPU centric collective implementations because the hardware used to form the offload unit can be specialized for the performed tasks, while a general purpose CPU has to deliver high performance for a wide range of codes.

2. Introduction

In this chapter we will describe our motivation for this work. We will explain why collective offload is an important research topic. An overview over the related work in that field will be given. We will provide background information about concepts and technology used in this work.

2.1. Motivation

Parallelism is steadily increasing in high performance computing platforms. This can be observed, for example, when looking at the Top500 list[1]. In the Top500 List there is an entry for the number of processors that each system consists of. In Figure 2.1 we give an overview over the development of the number of CPUs in the worlds most powerful supercomputers over time.

In parallel computing we can distinguish between two fundamentally different ways to communicate. Point-to-Point communication involves exactly two parties and data is transferred from one entity to the other. In collective communication a set with arbitrary cardinality of processes communicate. There are different types of collectives, for example a broadcast, where a data item from one process is replicated on all processes involved in the collective. Collectives provide a higher level of abstraction than point to point messages and should thus be favored where possible [Gor04]. The reasons fall in two different categories: One is the maintainability of the code: Collectives are easier to understand than their equivalent point to point message passing codes, they also need fewer lines of code. The other category is performance and the potential for optimization: As collectives are built on top of point to point operations they will automatically benefit from optimized point to point primitives (as the code that does not use collectives would), however, it is also possible to optimize the collectives implementation. Many people have worked on this topic in the past.

[1] `http://www.top500.org/lists/2011/06`

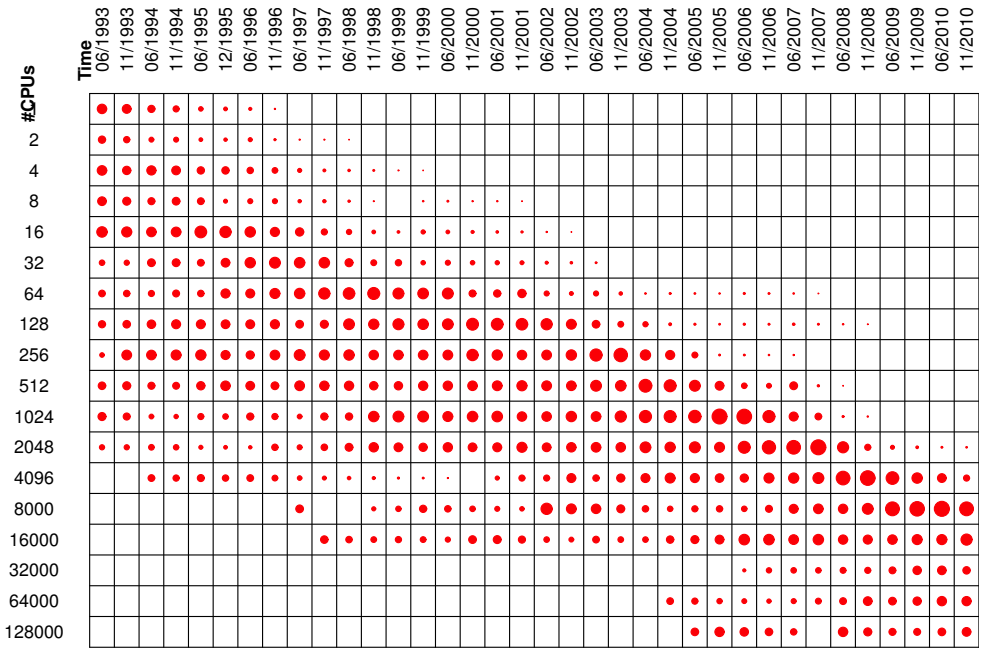


Figure 2.1.: Number of processors in the top 500 supercomputers over time. The size of each dot represents the performance share of the respective top500 list contributed by systems with the given number of processors.

One optimization for collectives that has been suggested by different people [HZ07, HGLR07a, AHA+05] is to overlap the communication time spent in a collective call with computation. This is based on the idea that most of the time spent in a collective function is lost waiting for messages, while the CPU is idle. Since collectives consist of multiple point to point messages, often with dependencies between them, such overlap is not trivial to implement, as the CPU is needed to progress the collective.

The two most common approaches to solve this is either using an extra progression thread or put the burden of progression on the user; in that case the user has to call a progression function in regular intervals during the computation that overlaps the communication. Both approaches have their advantages and disadvantages, [HL08] gives a good overview. The extra progression thread either overbooks one core, which can lead to imbalances in the execution times of MPI processes and contributes to the problem known as Operating System noise [PKP03] or it can be run on a spare core, which it might not utilize fully at all times. For the user (or application developer) it is hard to know the optimal interval at which he should call the progression function, especially since that depends on the system parameters, such as network latency and bandwidth, so it likely has to be adjusted for different clusters. Furthermore it can be impossible to manually call a progress function during a computation step, for

example if the computation step is actually a library call, for example to an FFT library and this library does not "care" about progression of collectives.

One possible solution could be the use of interrupts, generated by the networking hardware upon the arrival of data. However, such interrupts have a relatively high latency (about $1\mu s$ on a modern Linux system). This is one of the reasons most high-performance networks try to bypass the operating system where possible [SWP01] and rely on polling. However, for standard 10 Gigabit Ethernet is is possible to leverage interrupts for the progression of collectives, as we have shown in [SEHR11].

Another option to achieve the desired overlap of communication and computation which avoids most of the progression issues is moving as much as possible of the work needed to perform a collective communication away from the host CPU to a co-processor. This approach is called collective offload [BGRU06]. Offloaded non-blocking collectives not only help with progression, they can also mitigate some aspects of the problems associated with operating system and network noise [HSL10, HSL09b, HSL09a]. As shown in Section 2.3 collective offload started by providing hardware support for simple collectives such as Barrier and Broadcast, often implemented on the network processors that were already present in commodity or HPC network cards. However the implementation of the collective was not changeable by the user, the collective implementation was "hard-coded" into the network card. Recently this began to change, as Mellanox released the ConnectX-2 Infiniband Adapters [GPS+10] which offer hardware support for user specified collectives. While it has been shown that several important collectives can be implemented with ConnectX-2 efficiently [VGL+11, SKSP10] the interface of these adapters is backward compatible with old Infiniband hardware. Thus it is not trivial to understand which types of collectives can be implemented efficiently using that interface. In general collective functions can be modeled as dependency graphs of send and receive operations as well as local transformations on data. We are not aware of any proof or rationale which class of graphs can be handled by the ConnectX-Interface. Furthermore the actual design of the ConnectX-Interface is proprietary, so it is unclear which part of the collective execution is done in hardware and which is done in the driver. The Portals 4 API [Rie08] also offers functions that allow the user to specify collective functions in a dependency-driven manner. However, we are currently not aware of an actual hardware implementation that is capable of executing a collective function specified in that manner.

With this work we want to contribute to the field of collective offload for HPC networking by creating an architecture that is capable of executing arbitrary collective functions. Since the performance of such a design are hard to predict in a theoretical manner we evaluate the design by implementing it in a hardware description language on an FPGA.

2.2. Outline of this Thesis

This thesis is organized in five chapters. In the first chapter, the Introduction, we will motivate our work and by explaining why collective communication is an important research topic in high performance computing and why we believe that offloading is one of the branches of this topic that more work should be done in. We will describe the development in the area of collective offload and give some insights on the limitations of each approach previously taken. We will describe GOAL, a language to express collective communication patterns and EXTOLL, a high performance network based on HyperTransport which can be implemented using Field Programmable Gate Arrays. Since a large part of this thesis is about our prototypical GOAL interpreter, implemented on an FPGA, an introduction into to the concepts on which FPGA hardware is based upon concludes the introductory chapter.

In the next chapter we will talk about several problems that emerge when one tries to perform complete collective offload. The root of those problems is the fact that the memory available to a collective offload unit on a network interface card is much smaller than memory available to message passing frameworks that run on the host CPU. In addition to that the memory latency is typically larger for hardware on a peripheral bus than for the CPU. However, a collective offload unit which puts severe limits on the size of the executable collective communication schedules, compared with frameworks running on the CPU, is not of much use. The inventors of GOAL already thought about this problem and proposed a solution that involves splitting a large communication schedule in smaller parts which can be executed by the offload unit sequentially. We will show that this approach cannot be used in all situations. One of the things that take up a considerable amount of memory in traditional message passing frameworks is the buffering of unexpected messages. This is traditionally solved by using a rendezvous protocol which introduces more synchronization between the communicating peers to ensure no data is sent before the receiver is ready to receive the data into its final destination. Even though this approach does not buffer data it still needs to save state for every unexpected message. We show that it is possible to design point to point protocols without this property — in our proposed protocol unexpected messages can be discarded and are retransmitted when both peers are able to perform the message exchange. Because the new protocol is quite complex and we are unable to test the new protocol in practice we model the protocol in PROMELA and perform formal verification on that model.

All message passing frameworks (in contrast to communication frameworks designed around the partitioned global address space paradigm) need the ability to match incoming messages with preposted receives. Since the GOAL interpreter designed in

this thesis supports both, message passing and remote direct memory access we also need to provide such functionality. Traditionally message matching is implemented using dynamic data structures such as linked lists. Those data structures are relatively simple to implement on a host CPU where the operating system provides memory management and abstracts the details of the memory hierarchy away from the programmer. We benchmark the speed of traditional host CPU based matching and conclude that a hardware based matching unit can be faster (if running at the target clock frequency of 200 MHz) only if it is fully pipelined and performs one "matching comparison" during each clock cycle. We go on to explain how such a unit can be built in hardware.

In the next chapter a design of a GOAL interpreter unit is developed using the concepts of the previous chapter. The tasks of such a unit are analyzed to decouple parts of the execution and provide multiple sub-units that can be pipelined or work in parallel. The transceiver interface used by the GOAL Unit is introduced and the protocol is refined to contain all information necessary to start transfers using the transceiver unit. We describe how such a GOAL interpreter unit can be build in hardware and how scarce resources are shared between sub-units. The GOAL concept does not only define message passing functionality but also arithmetic operations on local data. We benchmark Xilinx IP cores that implement arithmetic operations for different data types and introduce a concept for an extension to the hardware GOAL unit.

In the last chapter we evaluate the performance achievable by our design when implemented on a Xilinx Virtex6 FPGA. We show that the complete schedule interpreter can be synthesized by standard tools to run at a clock speed of 212 MHz. We also give an overview over the resource requirements of each component our design contains. The overall design uses less than 10% of the Virtex6 XC6VLX75T-3ff784 FPGAs (the "biggest" FPGA available in the free version of the Xilinx ISE 13.2 tool we used for development) logic resources, while using less than 30% of the available BRAM blocks.

2.3. Related Work

Optimizing collective communication has been an important research topic for many years, as the scalability of many scientific applications is limited by the latency of such collective communication tasks. New communication schemes for collectives are proposed regularly [HK02], as the "optimal" communication scheme for a certain collective depends on many parameters such as the message size, the number of

processes involved and the network parameters such as latency, bandwidth and host overhead [VFD00].

One way to hide the latency of collectives is to overlap them with computation [HGLR07b]. Non-blocking collectives will also be included in future versions of the MPI standard [CGH94].

When using non-blocking collectives in MPI there are multiple ways to ensure progression. The most common options nowadays are manual progression, where the MPI application frequently has to call MPI functions such as MPI_Test to progress outstanding collectives and utilizing a separate progression thread. Both alternatives have disadvantages [HL08] — a promising third option is to offload the progression of collective functions into the network adapter [GPS+10]. This not only helps to solve the progression issue, it can also speed up the message passing framework as parts of its implementation can be carried out on specialized hardware instead of utilizing a general purpose CPU.

Doing so creates a new problem: How can collectives be encoded in hardware in such a flexible way that users can change their implementation freely and thus profit from the ongoing research efforts to design new collective communication schemes. Furthermore, collective should be implemented in a platform independent way, so that they do not have to be rewritten for every hardware platform?

We will continue with an overview over techniques used to offload collective communication from the host CPU to the network interface card.

2.3.1. NIC Based Packet Forwarding

The Barrier collective is the simplest collective for NIC offload since it does not communicate user data. The only information that has to be sent and received by processes is the information which process has already entered the barrier. Several algorithms [Höf05] exist which accomplish the transport of that information by means of sending and receiving zero-byte messages, i.e., the mere fact that a message was received from a particular peer conveys information about the state of possibly multiple other nodes. An example for such a Barrier implementation is a Binomial Tree Barrier:

When a new communicator is created, each rank in that communicator is assigned to exactly one node in a binomial tree with the same size as the new communicator, as in Figure 2.2. Now a barrier can be performed in the following way: The leaf nodes

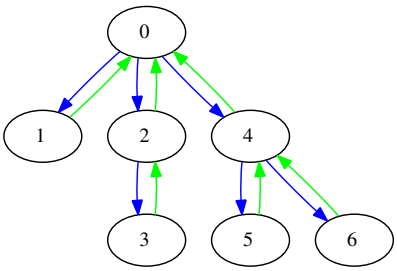


Figure 2.2.: Mapping of ranks to a binomial tree of size 7

that have no incoming green edges send a special "barrier-packet" to their parent rank (along the green edges) as soon as they enter the barrier. Each node waits until he received a barrier-packet for each incoming green edge. If rank 0 (the root) received all barrier-packets it will start the second round by sending another type of barrier-packet down the tree. When a node receives a second-round barrier-packet it passes it to its children and leaves the barrier.

To implement the barrier it is sufficient for each node to detect the barrier-packets and send them to the parent or child nodes, depending on the type of barrier packet received. Such a scheme can be implemented in the NIC firmware at a very low level. This was done for example for the barrier operation on Myrinet/GM in [BPS+01] and resulted in a considerable performance increase (speedup of 1.83 for eight nodes). A similar approach was chosen to implement an offloaded broadcast in [YBP03]. The main benefit of this offloading scheme is that it is really simple and therefore very low latencies can be achieved. The downside is that such simple schemes do not exist for all collectives. Also such implementations are quite unflexible. If the user wants to change the barrier implementation from a binomial tree barrier to a k-ary tree this can only be done by modifying the firmware of the network card. Also not all collectives can be implemented with such simple packet forwarding rules.

2.3.2. Hardware Barrier Implementations

Since the barrier collective does not transport any data, the latency of the network that is used to implement it determines its execution time, while the bandwidth of the network is less relevant. This makes the barrier collective different from all other collective operations which transport data. This discrepancy in requirements has led to the idea of utilizing a separate, dedicated network for barrier operations. It has been shown that, for a small node count, such a network can easily be implemented by using cheap commodity components such as an FPGA evaluation board and the

parallel port [HMMR06]. However, since the hardware requirements for such a barrier-network are low, this concept is also applicable, and has been used, in large parallel machines such as the Earth Simulator [HYK03] or IBMs Blue Gene/P [AHA+05]. However, also hybrid approaches have been suggested, where the barrier network uses the normal interconnection network but the network switches contain specialized hardware to speed up the barrier operation [SSP97]. All those approaches fall in the category of collective offload, as the host CPU is freed from the task to keep track of message dependencies.

2.3.3. ConnectX2 CORE-Direct Collective Offload Support

In recent years, various other low level network interfaces that strive to offer collective offload features have been proposed. For example ConnectX-2 CORE-Direct [SKSP10, GPS+10] introduces a *Management Queue* to the InfiniBand Architecture [Pfi01]: Traditionally InfiniBand connections are associated with a pair of queues. Each peer has a send and a recv queue. Both queues together are called *Queue Pair (QP)* These queues contain *Work Queue Elements (WQE)*. The host can add new Work Queue Elements to queue pairs, for example to initiate a send, and also poll a *Completion Queue (CQ)* to check if a particular operation was completed by the network card.

The idea behind the Management Queue is to provide hardware offload capabilities for a sequence of operations without modifying the current architecture. The Management Queue allows the user to specify additional operations: Send-Enable, Recv-Enable, Wait, Calculate. The enable operations instruct the HCA to enable the corresponding WQE (i.e., a send, recv or calculate operation). Queue elements which are not enabled yet, as well as all queue elements which are behind such an un-enabled element in a queue, cannot be processed by the network card.

Note that this is fundamentally different from GOAL: In GOAL executable elements will not be blocked by other elements, as long as the schedule is small enough to be processed at once. If a similar behavior is desired it would be necessary to create a new queue-pair for each network operation. However, Infiniband is specifically designed to keep the number of queue pairs as low as possible as queue pair creation is an expensive operation.

The Wait operation instructs the HCA to wait for a specified number of completion events before the following WQE are processed. The Calculate operation can be used to perform local reduction operations.

It has been demonstrated that both low-level interfaces can be used to specify collective communication operations. However, specified operations can hardly be optimized and transformed automatically in this representation.

2.3.4. Collective Offload Support in the Portals 4 API

In Portals 4 [Rie08] triggered operations have been added to allow offload of collective operations specified with the Portals API. These triggered operations support not only remote memory access (PtlTriggeredPut() and PtlTriggeredGet()) but also atomic operations (PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredSwap()). For each triggered operation, the user has to specify a counter and a threshold value. If the counter reaches the threshold the operation is triggered. Counters can be influenced by the user with PtlCTInc() but can also be influenced independently from the user by a triggered operation PtlTriggeredCTInc(). Most importantly a Portals memory descriptor (ptl_md_t) is capable of incrementing a counter when operations such as put and get, which operate on the corresponding memory descriptor, are completed.

2.4. Group Operation Assembly Language

In most MPI [CGH94] implementations in use today, such as MPICH [Gro02] or Open MPI [GWS06] collectives are built on top of point-to-point operations. Each collective implementation is a piece of (most often C) code that calls basic point-to-point or RDMA communication primitives. An example of such an implementation is given in Listing 2.1. In this simple barrier implementation each non-root node signals its arrival in the barrier by sending a zero-length message to root and then waits for another zero length message from root which signals the end of the barrier. The wait step is done by using a blocking receive operation.

The root process however uses non-blocking communication since he has to send and receive multiple messages. Non-blocking communication primitives allow the parallel processing of messages in this case. He explicitly waits for communication steps to finish using the ompi_request_wait_all() function.

Listing 2.1: Fragment of a Barrier implementation in Open MPI

```
1 /* All non-root send & receive zero-length message. */
  if (rank > 0) {
```

```
3       MCA_PML_CALL(send (NULL, 0, MPI_BYTE, 0, i, ...));
        MCA_PML_CALL(recv (NULL, 0, MPI_BYTE, 0, ... ));
5   }
    /* The root collects and broadcasts the messages. */
7   else {
        for (i = 1; i < size; ++i) {
9           MCA_PML_CALL(irecv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE,
                ..., &(requests[i])));
        }
11      ompi_request_wait_all( size-1, requests+1,
            MPI_STATUSES_IGNORE);

13      for (i = 1; i < size; ++i) {
            err = MCA_PML_CALL(isend(NULL, 0, MPI_BYTE, i, ..., &(
                requests[i])));
15      }
        ompi_request_wait_all( size-1, requests+1,
            MPI_STATUSES_IGNORE );
17  }
```

This way of implementing collectives is well suited for a host CPU centric message passing framework. However, it is rather difficult to use such an implementation for offloading the collective execution to a co-processor. As the collective is implemented in C, the co-processor itself would have to be able to execute (compiled) C code. The code above is also hard to understand because dependencies between send and receive operations are expressed in two different ways. In the non-root case the receive cannot start before the blocking send is completed. This is not explicitly stated, it is a result of the control flow in the program. In the root case the dependencies are made explicit using ompi_request_wait_all().

This makes it hard to reason about properties of collectives implemented in such a way. It is not clear for the reader if the implicit dependency in the non-root case is intended and necessary or not.

In [HSL09c] Hoefler et al. described the idea of expressing collective operations as a dependency graph between simple networking primitives, such as send and receive. They also included the possibility to specify transformations on process-local data. The idea to express interdependencies between simple operations instead of specifying the sequence of operations to be executed is an important concept in computer science, which finds manifestation in functional programming, dataflow-architectures and makefiles.

Hoefler et al. called the language that they used to express collectives in such a way the *Group Operation Assembly Language* (GOAL). They described a system where a collective is specified by constructing a dependency graph. This graph is called the GOAL graph or GOAL schedule. This graph is transformed into a binary representation during a compilation step. Such a compiled schedule can be executed by a GOAL interpreter unit.

In software such systems have been built before, for example in [SEHR11, NI10]. In this work we are developing an architecture for a GOAL interpreter hardware unit.

To give an overview over the capabilities of GOAL we will describe the API of the GOAL interpreter that has been built in [SEHR11] in detail in the following subsection.

2.4.1. GOAL API

The basic idea behind GOAL is to describe the dependencies among a series of communication and computation operations, so that the GOAL interpreter can execute these in any order which satisfies those dependencies. Dependencies can be described with directed acyclic graphs (DAGs). Such DAGs are the basic building blocks for GOAL.

So the first function which must be called to use GOAL creates a new GOAL_Graph object:

```
1 GOAL_Graph GOAL_CreateGraph()
```

Once we are finished with a GOAL_Graph (i.e. after it has been successfully compiled to a binary schedule) we can free all resources used by that graph with the function

```
1 int GOAL_FreeGraph(GOAL_Graph g)
```

We can add operations (send, receive and computations) to the GOAL graph. Those will be represented as vertices in the DAG. Of course each operation needs some parameters to specify its task.

In case of the send operation the user needs to specify the send buffer, the number of bytes to be sent and the destination rank.

```
1 GOAL_Vertex GOAL_Send(GOAL_Graph graph, void* buf, int count, int
      dest, int tag=0, GOAL_MemType mem=GOAL_USERSPACE)
```

The last parameter of this function is common for all buffers in GOAL: Buffers can be pointers to memory allocated by the program or byte offsets in the scratchpad memory region of that schedule. The buffer argument will be interpreted as a normal pointer if the "mem" argument is set to GOAL_USERSPACE (the default). We will explain the creation and usage of scratchpad memory in detail soon.

To receive data GOAL provides a receive operation, which can be added to the graph with the GOAL_Recv() function. The meaning of its parameters is analogue to GOAL_Send().

```
1 GOAL_Vertex GOAL_Recv(GOAL_Graph graph, void* buf, int count, int
      source, int tag=0, GOAL_MemType mem=GOAL_USERSPACE)
```

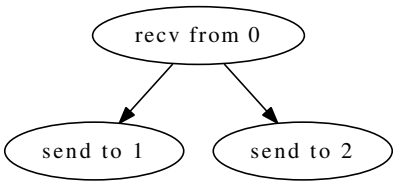
The third type of operations that can be used in GOAL (besides sending and receiving of data) are so called *local operations*. A local operation does not involve any communication, it is executed on the local rank (the one executing the corresponding schedule) only. The purpose of local operations (also called localops) is to enable the dependency based execution of simple arithmetic operations inside of GOAL. It must be ensured that local operations are "simple enough" to be executed by the GOAL interpreter. Therefore only predefined local operations are possible in the Basic GOAL API. The predefined operations available in GOAL are:

Datatype	GOAL_SINT					GOAL_UINT					GOAL_FLOAT				
Width	1	8	16	32	64	1	8	16	32	64	1	8	16	32	64
GOAL_MAX															
GOAL_MIN															
GOAL_ADD															
GOAL_SUB															
GOAL_DIV															
GOAL_MUL															
GOAL_COPY															
GOAL_AND															
GOAL_OR															
GOAL_XOR															
GOAL_WTIME	no type checking/conversion, puts timestamp [in s] in buf3 as 64 bit float														

Local operations can be added to a GOAL graph with the function:

```
1 int GOAL_LocalOp(GOAL_Graph graph, void *b1, void *b2, void *bres
    , GOAL_Op op, GOAL_DataType dt, int element_width, int
    num_elements, GOAL_MemType b1_mem=GOAL_USERSPACE, GOAL_MemType
     b2_mem=GOAL_USERSPACE, GOAL_MemType b3_mem=GOAL_USERSPACE)
```

As we mentioned earlier the execution of the specified operations will be dependency-driven. To add dependencies between operations the GOAL_Requires() function must be used. Each of the functions that will add an operation to the graph will return a GOAL_Vertex object which identifies that operation in the graph. These identifiers can be passed to GOAL_Requires() to link them together: Suppose we want to receive some data from rank 0 and upon reception we want to send this data to rank 2 and 3. The dependency graph for this case would look like this:



Using the GOAL_Requires() function

```
1 GOAL_Requires(GOAL_Graph g, GOAL_Vertex prereq, GOAL_Vertex
    target)
```

this can be expressed with the following piece of code:

```
1 GOAL_Vertex recv, send1, send2;
  recv = GOAL_Recv(g, &buf, 2, 0, GOAL_USERSPACE);
3 send1 = GOAL_Send(g, &buf, 2, 1, GOAL_USERSPACE);
  send2 = GOAL_Send(g, &buf, 2, 2, GOAL_USERSPACE);
5 GOAL_Requires(g, recv, send1);
  GOAL_Requires(g, recv, send2);
```

2.4.2. Scratchpad Buffer

Since the operations in a GOAL graph can be executed at any time after its definition by the user, possibly also multiple times, allocating and freeing temporary buffers is

difficult for the user. One might be inclined to do something like this to implement a tree based scatter:

```
buf = malloc(64);
recv = GOAL_Recv(g, buf, 64, 1, GOAL_USERSPACE);
send1 = GOAL_Send(g, buf, 32, 2, GOAL_USERSPACE);
send2 = GOAL_Send(g, buf+32, 32, 3, GOAL_USERSPACE);
GOAL_Requires(g, recv, send1);
GOAL_Requires(g, recv, send2);
free(buf);
```

However, the problem with this approach is that the buffer is freed in line 7 before the schedules execution is finished (in this example we did not even compile or start the schedule execution).

If we omitted free() in the last line this example would work fine, however we would have produced a memory leak as we never free buf now. One way around this would be to manually track the life cycle of the schedule resulting from the Graph g and free buf when the schedule is finished and will not be executed again. This is inconvenient and error-prone, especially in scenarios where we want to hide the actual collectives (i.e. scatter) implementation in a function call.

Therefore GOAL supports a very minimal memory management subsystem, the scratchpad buffer. We can inform GOAL that to execute the operations defined in "graph" we need at most "bytes" bytes of temporary buffer space with the function

```
int GOAL_AllocateScratchpad(GOAL_Graph graph, size_t bytes)
```

If we do that the GOAL interpreter will allocate a buffer of this size immediately after the schedule corresponding to "graph" is executed and that buffer will be destroyed as soon as the schedule's execution is finished. To use that buffer we have to supply offsets instead of pointers to any function that takes an argument of type GOAL_MemType. For the memtype argument we have to supply GOAL_SCRATCHPAD instead of GOAL_USERSPACE, which has to be used for standard memory accesses. The supplied offsets are relative to the start of the scratchpad buffer. Note that it is not possible to have more than one scratchpad buffers. If several buffers are needed the user has to allocate a single buffer large enough and perform the memory management himself.

The correct version of the example given above would look like:

```
GOAL_AllocateScratchpad(g, 64);
recv = GOAL_Recv(g, 0, 64, 1, GOAL_SCRATCHPAD);
```

```
send1 = GOAL_Send(g, 0, 32, 2, GOAL_SCRATCHPAD);
send2 = GOAL_Send(g, 32, 32, 3, GOAL_SCRATCHPAD);
GOAL_Requires(g, recv, send1);
GOAL_Requires(g, recv, send2);
```

To copy data to/from the scratchpad memory we can use a local operation of the type GOAL_COPY. See the documentation on local operations for more information.

2.4.3. Schedule Execution

To start the execution of a schedule the user has to call

```
GOAL_Handle GOAL_Run(GOAL_Schedule sched)
```

To check if a schedule's execution is finished, GOAL provides two different functions:

```
int GOAL_Test(GOAL_Handle handle)
```

will return 1 if the corresponding schedule (the one for which handle was returned when it was started with GOAL_Run) is finished executing, otherwise 0. It will not block

```
GOAL_Wait(GOAL_Handle handle)
```

on the other hand will block until the corresponding schedule is finished.

The algorithm used to execute a GOAL schedule works as described in Listing 3: All operations that do not have unmet dependencies are put in a Queue. Each element of that queue is executed as soon as possible. The execution however is non-blocking.

If the execution of an element a is finished, for example because data arrived that matches a previously executed receive operation, the callback function resolve_deps() is called. This function checks if there are operations in the graph that now, after a is finished, have no more unmet dependencies. If such elements exist they are put in the queue.

Algorithm 1: Schedule execution Algorithm, Startup

Input: Process-local GOAL graph G as:
a set of operations (A),
a set of dependencies ($D \subseteq A \times A$)

```
1 Q ← ∅
2 forall a_i ∈ A do
3     if ∀x : (x, a_i) ∉ D then
4         Q ← Q ∪ a_i
5 while Q ≠ ∅ do
6     a ← pop(Q)
7     execute(a)
```

Procedure `resolve_deps`(*a, G, Q*)

Input: Operation $a \in A$,
process-local GOAL graph $G = (A, D)$,
Queue of outstanding operations Q

```
1 forall (a, x) ∈ D do
2     D ← D \ (a, x)
3     if ∀y : (y, x) ∉ D then
4         Q ← Q ∪ x
```

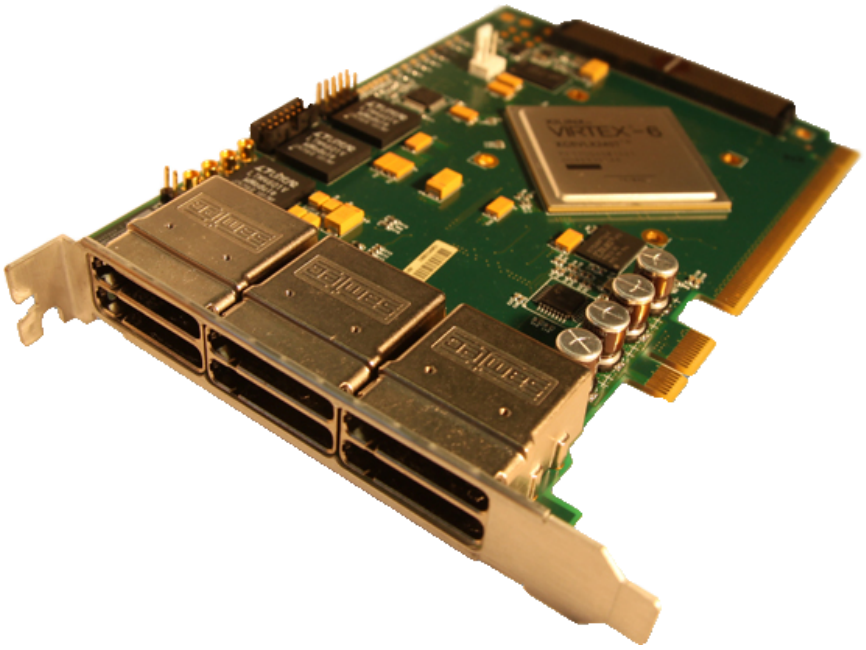


Figure 2.3.: The EXTOLL NIC, based on a Virtex6 FPGA

2.5. The EXTOLL Network

The EXTOLL network [NGFB09] is developed by the computer architecture group of Prof. Brüning at the University of Mannheim. It is a switchless high performance interconnect which utilizes a three dimensional torus network topology. Current versions achieve point to point latencies around 1 μs which is comparable to well known high performance interconnects such as Infiniband. Currently available versions of EXTOLL are based on Virtex6 FPGAs and leverage HyperTransport as their host interface. This makes EXTOLL an excellent platform for high performance networking research as researchers can utilize the HyperTransport core [SGB07] and hardware developed by Prof. Brüning's group to prototype their own network card designs [NFG+]. Figure 2.3 shows a picture of the current version of the EXTOLL NIC.[2] — the Vertex6 FPGA and the six links necessary to form a three dimensional torus network can be seen.

The EXTOLL network uses an on-chip network called HTAX to connect the HyperTransport core to the other functional units of the card. The most notably of these are the VELO and the RMA unit. The VELO unit offers reliable low latency point to point message transfer. The RMA unit provides get and put access to remote

[2]Source: `http://www.extoll.de/`, Copyright: EXTOLL GmbH

memory, as well as atomic operations. Our GOAL interpreter is designed to use both message transfer modes simultaneously. For small command/signaling packets to remote GOAL interpreters VELO messages should be used, while RMA is leveraged for transferring data from and to application buffers. The interfacing of the EXTOLL unit however is outside of the scope of this work — another thesis which is also done at the computer architecture group of Prof. Rehm at Chemnitz University of Technology simultaneously with this work will develop an abstract DMA based interface which we will use. This interface is described in more detail in Section 5.2.

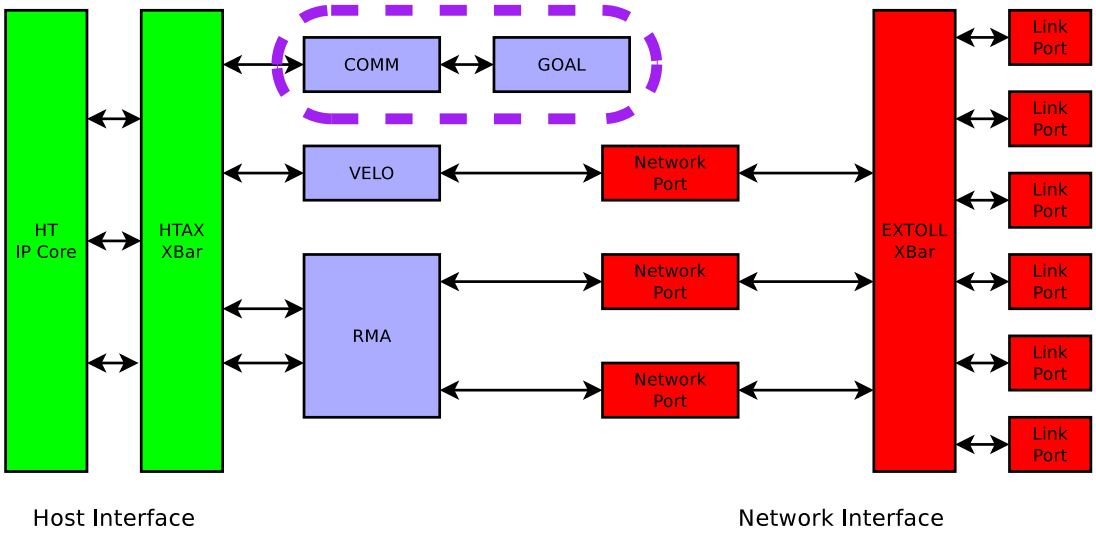


Figure 2.4.: EXTOLL Block Diagram, including the proposed new units

The general structure of the EXTOLL network, including a proposal on how to integrate the GOAL unit and the abstract communication interface described before, which we will call COMM-unit from now on, is given in Figure 2.4. The GOAL unit is only connected with the COMM unit. The COMM unit is connected to the HTAX crossbar, similar to the VELO and RMA unit. The COMM unit is not connected directly to the network components as it does not communicate with remote entities by itself, it uses the VELO and the RMA unit to do so.

2.6. Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are integrated circuits that can be configured by the user after manufacturing. That is why they are called “field programmable” — their functionality is defined by the customer “in the field” instead of the manufacturer. FPGAs can implement any logic function that an application specific

integrated circuit (ASIC) could perform. The configuration of an FPGA is usually specified in a hardware description language (HDL) such as Verilog or VHDL. Similar tools are used for ASIC design. Therefore, and due to their reconfigurable nature, FPGAs are well suited for the prototyping of ASIC designs.

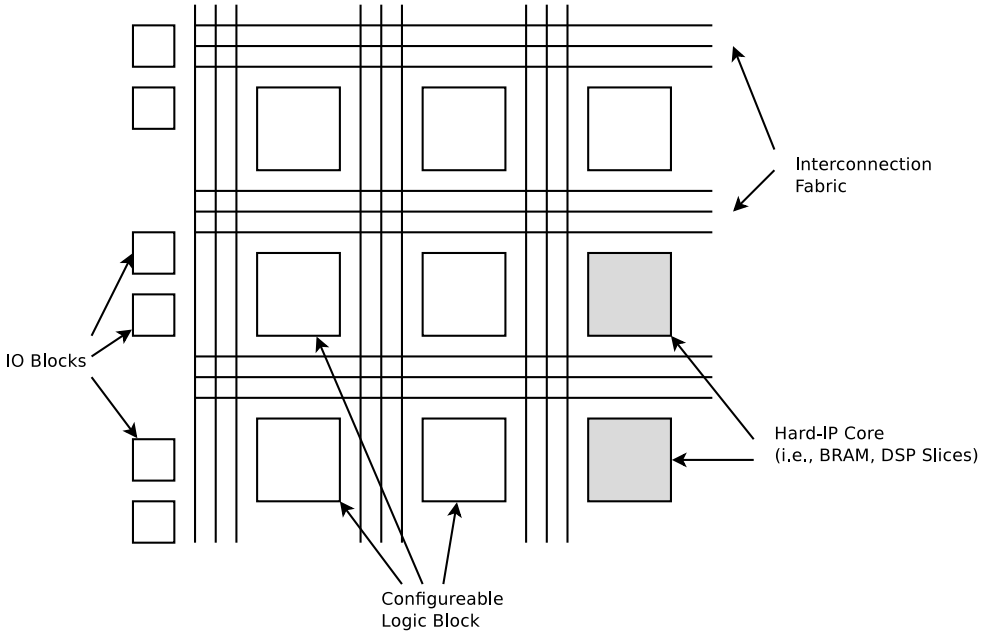


Figure 2.5.: Structure of an FPGA

FPGAs consist of IO pads, configurable logic components blocks (CLBs), and a hierarchy of reconfigurable interconnects that allow the blocks to be connected together. Logic blocks can be configured to perform complex combinatorial functions, or merely simple logic gates. In most FPGAs, the logic blocks also include memory elements, such as flip-flops. Many FPGAs also include direct hardware implementations of often used basic building blocks, such as memory, arithmetic units that are useful for digital signal processing or even CPUs such as a PowerPC core. These *Hard IP Cores* are not relocatable on the FPGA, as they are directly implemented in silicon. FPGA manufacturers and third parties also provide *Soft IP Cores*. Those are synthesizable modules that can be placed on an FPGA when needed. Examples for such cores are the MicroBlaze [Inc11d] processors offered by Xilinx.

In the following we will explain how FPGAs achieve the flexibility to implement arbitrary logic functions. We will not explain the architectural details used in a specific FPGA of a specific vendor, as the details vary to some degree. We will rather explain the high level concepts behind them, as a good understanding of those is necessary to understand the limits of the usefulness of FPGAs for ASIC prototyping. A more technical and hardware specific introduction is given in [BR96].

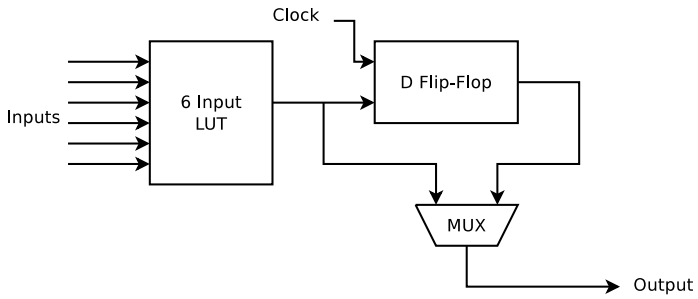


Figure 2.6.: Structure of a CLB Slice

The general architecture of FPGAs is shown in Figure 2.5. The configurable logic blocks usually consist of one or more slices or cells. Each cell can be used to store a small amount of data or implement a logic function over a small number of input bits. An example of such a slice is given in Figure 2.6. The LUTs can be thought of as memory with an address width equal to the number of LUT inputs and a data width equal to the number of LUT outputs. Any logic function over the inputs can be realized by directly writing the truth table for the output bit into the LUT. The FPGA interconnect is also configurable. Each point where interconnect lines are crossing in Figure 2.5 can be thought of as a small crossbar switch. Of course in real FPGAs each manufacturer introduces their own variations and optimizations to the basic architecture described here.

The fundamental architecture of FPGAs has some important implications that have to be considered when designing for FPGAs. The fact that the design is mapped to relatively simple functional units implies that the capabilities of those units have to be considered during the design. For example when memory units are implemented the synthesis tool (i.e., the "Verilog to FPGA" compiler) will map the implemented memory to the flip flops in the CLBs or it will utilize the memory blocks that are available in silicon. This mapping can be constrained by the designer, we can instruct the synthesis tool to use only a specific type of memory blocks. If the requested memory is larger than a single block, the synthesis tool also will implement addressing logic which will be implemented using CLBs. Each entity in the signal path will introduce a certain delay in that path. It can be differentiated between delay introduced because of the signal routing from one CLB to another and the delay introduced by the logic gates in the path. The sum of all delays of the first kind is called *routing delay* or *net delay*, the sum of all delays of the second kind is called *logic delay*. The maximum clock frequency achievable by an FPGA design is bounded by those delays (note that there are also other kinds of delays which are not described here, such as *setup time* and *hold time*). The synthesis tool will ensure that the clock period is larger than the time needed for the signal to propagate from the signal source to its sink. Examples for signal sources are input pins or registers, while

signal sinks can be output pins or also registers. A direct consequence is that we have to pay attention to the signal path lengths during the design phase. A signal path can be shortened by adding registers where the signal value is temporarily stored and propagated further in the next clock cycle. Of course this results in the signal needing more clock cycles to reach its final destination, however it also allows a higher clock frequency for the overall design. Therefore such buffering registers should not be introduced arbitrarily but only in the critical (longest) paths of the design.

The Xilinx ISE design tool is capable of showing detailed information about the placement of design on the FPGA and also gives some information about the critical path. Figure 2.7 shows some of the BRAM blocks used in our design (large green rectangles) and the address decoding logic necessary to utilize them as a bigger block of memory with the connections between elements shown as white lines.

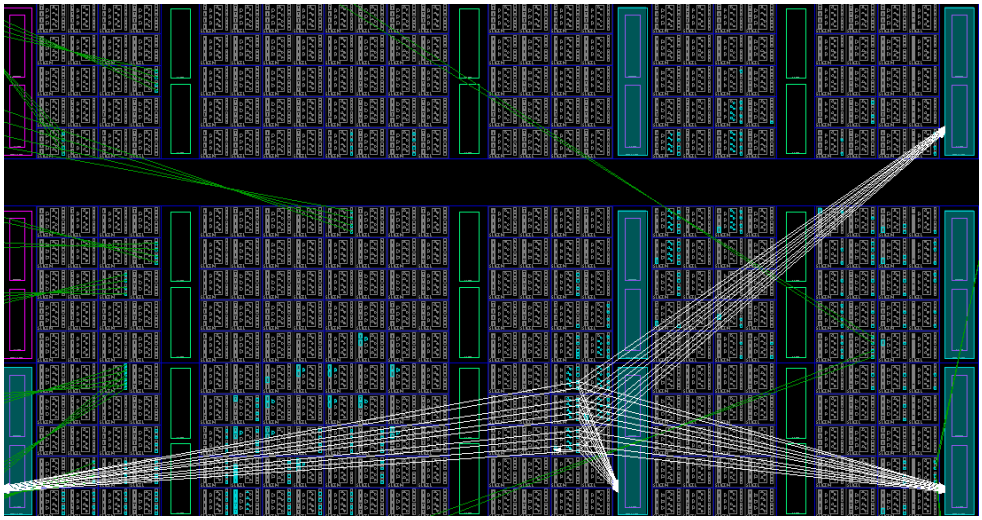


Figure 2.7.: PlanAhead Screenshot

3. Dealing with Constrained Resources

In this chapter we will first describe the limits that are imposed by the prototyping platform. We will follow up with an explanation of some basic building blocks for collective communication functions. We will analyze some theoretical properties of the graphs that result from translating those building blocks into GOAL. We will revisit some of the statements made in the original GOAL paper [HSL09c] about how arbitrarily large GOAL graphs can be executed with limited memory resources. We will then propose solutions to deal with some of the identified problems.

3.1. Hardware Limitations

The hardware chosen as reference platform contains a limited number of dual-port block ram which allows read and write access in a single clock cycle. The XC6VLX75T-3ff784 provides 312 18 KiB blocks and 156 36 KiB blocks [Inc11e]. Parts are already used by other components like the HT core and the DMA unit which is used for communication with other nodes or the host memory.

After talking with the EXTOLL developers, who have experience in FPGA and ASIC design it became clear that our design should be limited to using around 2 Mbit of simple dual port ram for all components developed in this thesis. Simple dual port ram differs from the true dual port ram available on the Virtex-6 FPGAs. Simple dual port ram, or two port memory, offers one read and one write port. Both ports can be used simultaneously. True dual port ram also has two ports, however, each port can be used as read and write port.

Beside the memory on the FPGA, the host memory can be utilized. This would introduce extra latency for reads and writes. On a Iwill DX8-HTX a latency of 390 ns for 8 Byte reads and 65 ns for 8 Bytes writes were measured using a DMA engine on the FPGA [Vol10]. Because of these relatively high latencies and the fact that utilizing the host memory during collective execution would slow down the overlapped computation significantly we decided against using the host memory as “swap space” for our GOAL interpreter unit early on in the design of our architecture.

3.2. Common Collective Functions in GOAL

As explained in section 2.4, the GOAL schedule consists of operations and dependencies between them. All information about these elements must be stored inside the binary format and maybe rearranged for better usage of the units of the hardware interpreter. The difference between send and receives are ignored, because they are only endpoints for the same transfer and do not show any significant difference in terms of memory footprint inside the schedule.

It could be required that the complete graph of all local schedules needs to be processed to determine different properties of an algorithm. For example detection of group operations with barrier semantics can be done and the user implementation could be replaced by a more efficient version for the current hardware architecture and network topology. The most important property that can be checked with the global graph is the loop-freeness of all operations to prevent global deadlocks.

For the design of the schedule interpreter unit the most important properties of the schedules are the maximum size of a process local schedule for each collective. This number bounds the maximum number of ranks that can be used for each collective. The rank with the largest GOAL schedule is usually the root rank for group operations where everyone sends directly to the root rank like Linear Gather and Linear Sync Gather. There are also group operations where the size of the schedule is roughly the same on all ranks for group operations like Bruck Barrier, Recursive Doubling Barrier and Pairwise AlltoAll. Tree based algorithms like Binomial Tree Gather, k-ary Tree Barrier and k-ary Tree Bcast tend to have the biggest schedule size for a child of the root rank. The extra data compared to the root rank schedule comes from the extra dependency of receives which must be finished before the send to the child or the root rank can be started.

An additional aspect is the out-degree of an operation. It describes the number of operations which may start after a specific operation has been finished. A special exception are the independent operations which can be started directly after the schedule was started.

Binomial Tree Gather

This Gather implementation sends the data to the root rank along a binomial tree. Each rank that is not a leaf or root rank will send all its data to a

temporary buffer and forward this data to the father after all children also send him their data. The root rank will transfer his data directly to the target buffer.

This means that we have $\lceil 2^{\lceil \log_2 p \rceil - 1} \rceil - 1$ ($\approx \frac{p}{2}$) non-leaf ranks without the root that have to wait for the copying transfer to the temporary buffer and $p - 1 - \lceil \log_2 p \rceil$ children without the children of root have to finish their transfer before the father rank can start his send to his parent.

The rank with the most dependencies is a rank directly below the root because it must ensure that the data from all children was already received before it can be re-transmitted to the father rank. This rank has $\lceil \log_2 p \rceil - 1$ children and an extra transfer from the local buffer to the destination buffer has to be made.

The maximum out-degree is one for all communication with more than one rank. The send operation is the only operation that has to wait for other operations to finish and therefore the maximum number of adjacent operations is one for all explicit operations. The root rank will also start a receive and send operation to copy the own send buffer to its destination. Each rank will also start a receive for all children. The root rank has $\lceil \log_2 p \rceil$ children and will execute $\lceil \log_2 p \rceil + 2$ operations directly after the schedule was started.

global Sends/Recvs	$2(p + \lceil 2^{\lceil \log_2 p \rceil - 1} \rceil - 1)$
global Dependencies	$p + \lceil 2^{\lceil \log_2 p \rceil - 1} \rceil - \lceil \log_2 p \rceil - 2$
maximum Sends/Recvs	$\lceil \log_2 p \rceil$
maximum Dependencies	$\lceil \log_2 p \rceil - 1$
maximum Outdegree	$\lceil \log_2 p \rceil + 2$

Bruck Barrier

Each rank in a Bruck Barrier [BHK+02] uses $\lceil \log_2 p \rceil$ rounds in which a rank sends to another rank a message and receives from one rank. A message informs the receiver that all messages from earlier rounds have been received by the sender. This makes it necessary that each receive and send of round i depends on the receive of the round $i - 1$ to prevent too early propagation of information.

Each participating rank in a Bruck Barrier has the same amount of operations and dependencies. The global amount can be divided by p to get the local operation and dependency count.

Each round has to wait for the previous round before the send and receive are started. The last receive is used as it contains the implicit information about

the previous rounds on the sender rank. The maximum number of adjacent operations can be found on a receive which is not part of the last round.

maximum Sends/Recvs	$2(\lceil \log_2 p \rceil)$
maximum Dependencies	$2(\lceil \log_2 p \rceil - 1)$
maximum Outdegree	2
global Sends/Recvs	$2(p * \lceil \log_2 p \rceil)$
global Dependencies	$2p(\lceil \log_2 p \rceil - 1)$

RecDbl Barrier

Recursive doubling barrier uses a similar number of sends and receives like Bruck Barrier, but uses only a single neighbor in each round. This creates the restriction that for r rounds only $p = 2^r$ ranks are used. This is necessary because in each round the neighbor rank is calculated by flipping in round i the bit i of our local rank identifier.

The algorithm was slightly changed to allow also $p \neq 2^r$ by letting all ranks with identifiers not smaller than 2^r to announce themselves to ranks smaller than 2^r. The ranks smaller than 2^r will inform after round r that all other ranks were already inside the barrier to the ranks not smaller than 2^r.

Our RecDbl Barrier implementation has the same amount of operations and dependencies for all $p = 2^r$, but extra dependencies and transfers for ranks which must communicate with other ranks identified using values larger than 2^r in situation where $p \neq 2^r$.

The maximum number of adjacent operations can be found on receive operations that are not participating in the last round. The round based synchronization works similar to the Bruck's algorithm and has the same implications.

maximum Sends/Recvs	$2(\lceil \log_2 p \rceil)$
maximum Dependencies	$2(\lceil \log_2 p \rceil - 1)$
maximum Outdegree	2
global Sends/Recvs	$2(p * \lceil \log_2 p \rceil)$
global Dependencies	$2p(\lceil \log_2 p \rceil - 1)$

k-ary Tree Barrier

The k-ary Tree Barrier has a contraction and information phase. During the contraction phase each child in a k-ary Tree sends to his father rank that he entered the Barrier when all direct children also informed him that they entered the Barrier. The information phase ends when $p - 1$ messages were transferred and the root rank received data from all of his children.

In the information phase each rank sends to all direct children a message when he received a message from his father. Only the root rank will send a message to his children when he also received a message from all of them to start the information phase. The implementation uses a local send and receive on the root rank to reduce the difference in the implementation when switching from contraction to information phase and results in p messages in the information phase.

The dependencies on all ranks which have children are similar. These only have to wait in the first round for there children before the father can be informed and in the second round for the father to inform all direct children. The father for root is defined through the local transfer as his own father. This means that for each phase the total number of children in the tree is $p-1$ which is also the number of dependencies required.

A rank participating in a k-ary tree barrier waits for messages from its children before he informs his father rank. All receive operations have only a single send as adjacent operation during this contraction phase. The ranks will wait during the information phase for a message from their father rank. A single receive has a maximum of k adjacent send operation when more than k ranks are participating.

maximum Sends/Recvs	$2(\min(p,k)+1)$
maximum Dependencies	$2k$
maximum Outdegree	$\min(p-1,k)$
global Sends/Recvs	$(4p-2)$
global Dependencies	$2p-2$

k-ary Tree Bcast

The buffer of the root rank in an balanced k-ary tree is transferred to all direct children. This means that all ranks which are not root will receive a message from their father ranks. The total number of children in the tree $p-1$ is the amount of transfers needed. All inner ranks except the root rank must wait for the receive of the message from their father before the message is sent to all children.

A rank participating in a k-ary Tree Broadcast has the maximum number of operations when he has the maximum number of children k and he has to receive from his parent before he can retransmit it to them.

The information phase of the k-ary tree Barrier has the same structure as the k-ary tree bcast, but does not transport actual data during this phase. The maximum number of adjacent operations of a receive is the same.

maximum Sends/Recvs	$\min(p, k) + 1$
maximum Dependencies	k
maximum Outdegree	$\min(p - 1, k)$
global Sends/Recvs	$2(p - 1)$
global Dependencies	$p - k$

Linear Gather

During a Linear Gather, every rank sends his data to the root rank. The root rank only has to start a send for his own data and a receive for all ranks including itself to copy the data to the receiving buffer. There are no explicit dependencies needed to guarantee the correct order of all transfers because all transfers are independent.

The rank with the maximum number of operations is always the root rank because it has to transfer the data from the source buffer to the destination buffer and receive from all other ranks.

The linear gather has no dependencies between the operations and therefore only independent operations exist. The implicit start operation would have all operations as adjacent operations. The root rank would start a receive for all other ranks. Also a receive and send operation is used to copy the own send buffer to its destination.

maximum Sends/Recvs	$p + 1$
maximum Outdegree	$p + 1$
global Sends/Recvs	$2p$

Linear Sync Gather

Linear Sync Gather works similar to the Linear Gather, but each non-root rank sends an initial segment to the root rank. This segment is acknowledged by the root. The second segment is transferred to the root rank after the acknowledgment message was received. This increases the number of transfers for all buffers to 3 for every buffer excluding the source buffer of the root rank which is still transferred using a single send and receive.

The root rank has to add dependencies to the receive of the initial segments before it can send the acknowledgment message and the non-root ranks have

to add dependencies to prevent the transfer of the second segment before the initial segment was started and the acknowledgment message was received.

The root rank is also the rank with the maximum number of operations for Linear Sync Gather. The root rank has to copy all data from the source buffer to the target buffer, but must receive the initial segment of all other $p-1$ ranks and send a acknowledgment back to them before they start to send the remaining data to the root.

The root rank posts the receive operation for both segments and the send of the sync message for all other ranks after the schedule was started. The copying of the own send buffer to its destination will be triggered after all receives have been finished.

maximum Sends/Recvs	$3p-2$
maximum Dependencies	$p-1$
maximum Outdegree	$3p-3$
global Sends/Recvs	$2(3p-2)$
global Dependencies	$3(p-1)$

Pairwise AlltoAll

Pairwise AlltoAll uses rounds where each rank sends its data to another rank and receives data from a rank. $p-1$ rounds are necessary to finish the complete exchange of buffers. Each rank has to use a dependency to prevent the sending of data before the receive of the previous round has finished. Each rank also has to transfer its own send buffer to its destination buffer using a send and receive operation.

Pairwise AlltoAll has the same operations everywhere and the number of global operations can be divided by the number of ranks p.

Each rank starts a receive and send operation to copy the own send buffer to its destination. Each exchange round uses one operation to send a buffer and one to receive another buffer from a rank. Each send operation depends on the receive from the previous round, but all receive operations are started at the same time.

maximum Sends/Recvs	$2p$
maximum Dependencies	$p-2$
maximum Outdegree	$1+p$
global Sends/Recvs	$2p^2$
global Dependencies	p^2-2p

To validate above theoretical models of the described collective functions we assume the resource requirements for storing operations and dependencies for the new GOAL interpreter unit will be the same as they are for the in-kernel GOAL interpreter ESPGOAL [SEHR11]. Based on this assumption we are able to generate the binary schedules for the mentioned collective functions with our ESPGOAL implementation and empirically determine some of their properties, such as their size. Only send buffers with the size of 1 Byte were used during the tests.

Figure 3.1 shows that the memory usage per process for some collective communication algorithms grows logarithmically with the number of ranks involved, as predicted by our estimations above. Those algorithms distribute the operations over many ranks like Bruck Barrier [BHK+02] and Recursive Doubling Barrier. Binomial Tree Gather's number of operations and dependencies grows logarithmically with the number of ranks, but the number of scratchpad buffer grows quadratically with the number of involved ranks. There are also algorithms for which the schedule size is is determined by the number of outgoing edges per rank like k-ary Tree Barrier and k-ary Tree Broadcast.

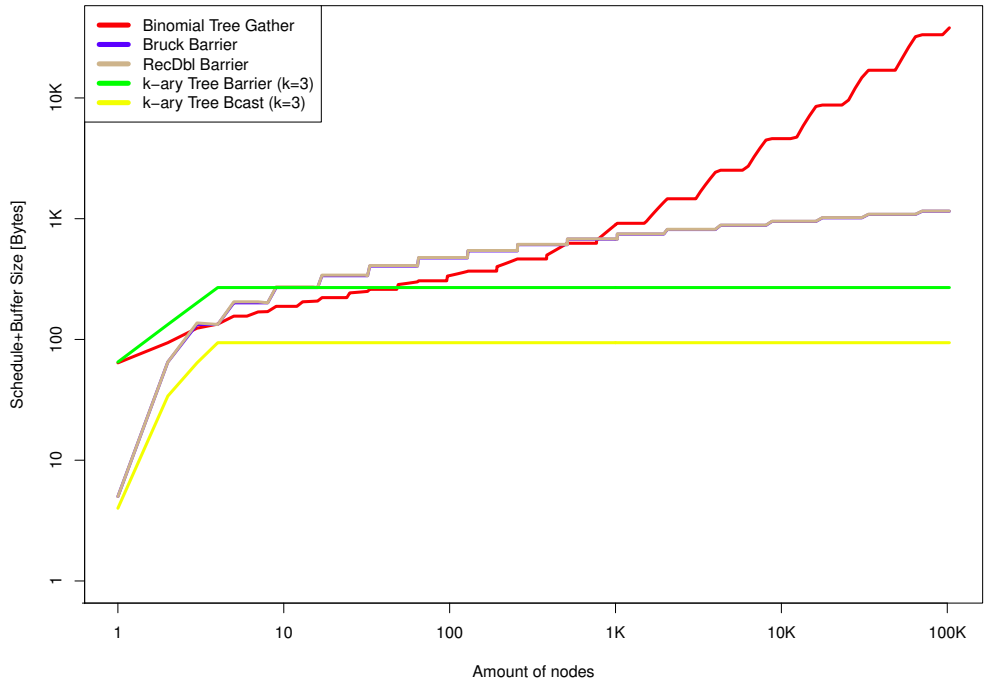


Figure 3.1.: Maximum schedule and buffer size for bounded or logarithmic growing schedules

Linear Gather and Linear Gather Sync have small schedules for nearly all ranks, but the schedule for the root rank grows linear with the number of ranks as shown in

Figure 3.2. The Pairwise AlltoAll is the only implemented collective operation with an linear growth of the schedule over all ranks.

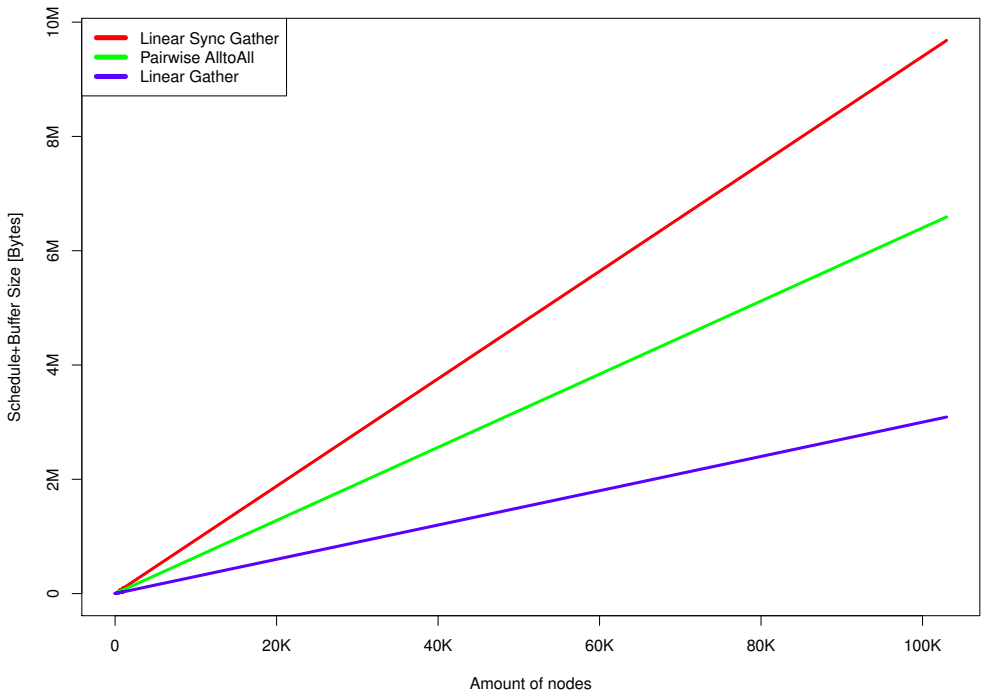


Figure 3.2.: Maximum schedule and buffer size for schedules growing linearly with the number of involved ranks

It can easily be seen that the Block RAM available on the FPGA is not be sufficient to store the complete schedule and the temporary buffers for some tested collective operations. We can only ensure that specific implementations with hard size limits will fit inside the available memory. Collective operations which grow with the number of ranks will automatically overrun any limit when enough ranks are used.

There are different ways to work around this limitation. One idea is to use the host to split the schedules in smaller parts [HSL09c].

The number of operation and dependency show the same characteristics which were also observed in Figure 3.1 and Figure 3.2. The only difference in Binomial Tree Gather can be explained through the size of the scratchpad buffer which was not added to this summary.

The maximum out-degree for operations shows similar characteristics as the maximum number of operations. There seems to be a group of collective communication operations like Bruck or Recursive Doubling Barrier that start multiple operations

with different types in each round. A schedule interpreter unit implementation can use this knowledge and provide separate processing units for parallel execution of the different operation types.

3.3. Schedule Representation for the Hardware GOAL Interpreter

In our previous GOAL interpreter implementation, ESPGOAL [SEHR11], the layout of the binary schedule memory was designed to keep all necessary information for the schedule interpretation in the same memory location and therefore increase the amount of cache hits during the interpretation of the schedule. The hardware implementation does not have the problem that access to one BRAM block should follow special patterns to increase the throughput or reduce the latency.

When more than one block of BRAM is necessary to store information, additional logic and register before and after the blocks are necessary to provide a single address space for the units which want to use the memory. The address bits must be used to decide which BRAM has to receive the write signal and from which BRAM the read result is taken. Read operations on the used FPGA take at least one cycle before the result is visible on the dout port of the BRAM [Inc11f]. Therefore, a register must be used to decide which BRAM block was addressed during the last read access. The additional logic and the distance between the not arbitrary placeable BRAM blocks increase the amount of time it takes to transfer the signal from the dout port of the BRAM to the input port of the next unit.

Many functional sub units need access to the data stored in the schedule during the execution of the schedule, but simple dual port memory has only a single write address and a single read address port. An arbiter unit has to be used to provide more virtual ports to other units. Read or write accesses have to be prioritized by the arbiter and the units have to be informed by the arbiter whether their last operations was accepted or rejected. Additional logic to decide which unit won and to store this decision for the information phase in the next cycle are necessary. This creates additional overhead which reduces the maximum speed of the units in hardware.

Those problems can be reduced by splitting the global address space and provide specialized memory blocks for different units. Each unit with private data can use a smaller block to store that information and therefore remove the need for additional

logic and makes it unnecessary to add wait cycles in situations were another unit were prioritized.

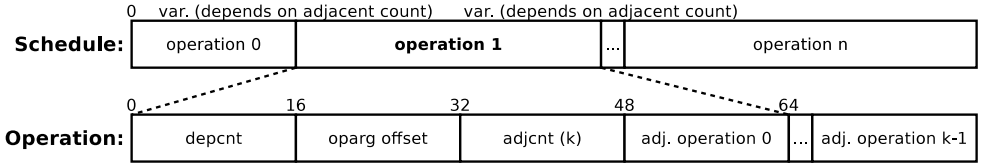


Figure 3.3.: Proposed Schedule binary representation

The unit which searches for finished operations in the schedule only needs the dependency counter, the number and list of adjacent operations to find dependent operations and to manage the number of operations which have to be finished before another operation can be started. The unit which controls the execution of the operations also needs the information were arguments are stored. This field can be interpreted as pointer in another address space which only stores operation specific data.

Figure 3.3 shows the schedule representation with all necessary 16 bit data fields and the additional pointers to the k adjacent operations in the schedule memory address space. The operation 0 at address 0x0 has a special function and will not be executed. The starting unit has to mark this operations as finished and all adjacent operations will get the dependency counter decremented. Operations which reach a dependency counter of zero will be started directly after the schedule was loaded. Using that special operation, the independent actions in ESPGOAL can be simulated without going through the whole schedule and searching for operations which already have a dependency count of zero.

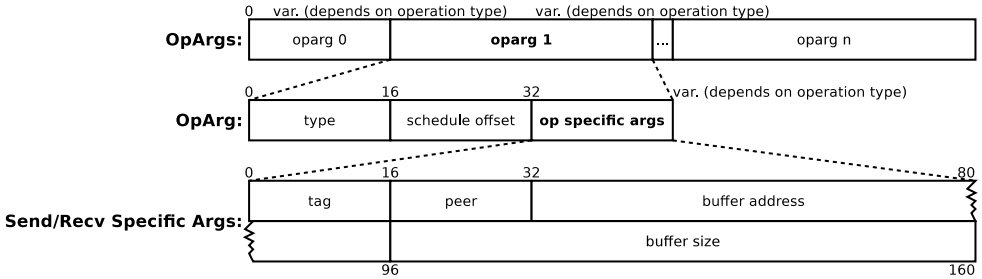


Figure 3.4.: Proposed Schedule operation arguments representation

The oparg offset points to an oparg element in the operation argument memory as shown in Figure 3.4. There is no actual connection between the oparg number and the position of the operation in the schedule memory. Therefore, each oparg element has to store a pointer to the operation in the schedule memory that was used to start

the operation and will be used to search for operations which can be started after this one finished. The interpretation of the argument and the size of it depends on the type which is stored in the fixed length operation argument header.

The only types and argument formats defined in this thesis are no-operation, RDMA get, send and receive. No-operation has no further arguments and therefore no operation specific argument format defined. Both send and receive share the same argument structure as shown in Figure 3.4. A RDMA get is a one sited operation which also needs the information about the buffer position of the remote buffer, but can drop the tag field which is only necessary for the matching during a send or receive. The complete example for an RDMA get operation argument can be seen in Figure 3.5.

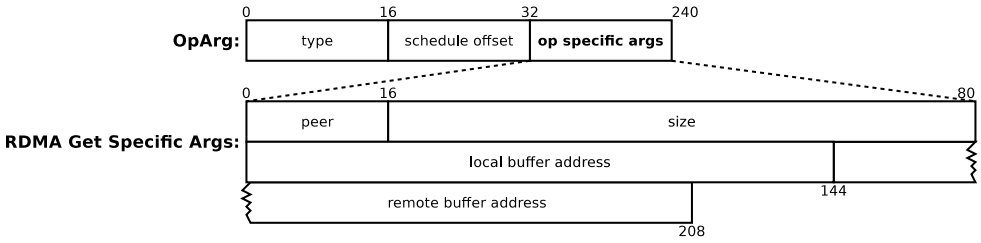


Figure 3.5.: Proposed Schedule operation argument representation for RDMA Get

Both memory address spaces are implemented with their own independent group of BRAMs. A size for the schedule operation memory has to be chosen which is small enough to store all arguments in the operation argument memory. An upper limit is the size of 2 Mib for the complete schedule interpreter unit. Therefore, not more than 1 Mib should be used for one memory region to allow all other memory regions to use their complete addressable range.

The simplest operation consists of a dependency counter, oparg offset and the adjacent counter. Three 16-bit words are necessary to store all information and a 128 Kib large memory can store 2730 operations for the smallest operation representation. The largest argument type as shown in Figure 3.1 is a RDMA get with 15 16-bit words for the complete entry. 40950 16-bit words would be necessary to store all arguments for that type and therefore 16 bit addresses would have to be used. The memory region which can be addressed needs a 1 Mib memory block.

It is also possible to only provide enough memory for smaller operations and let the generator of the binary schedule check that never more memory for the arguments are used than available. For example 32760 16-bit words are necessary to store the arguments for 2730 send or receive operations. Only a 512 kib large memory block is necessary to store the arguments. This could also heavily influence the placement of

Operation	Size [16-Bit Words]
NoOp	2
Send	12
Recv	12
RDMA Get	15

Table 3.1.: Size of operation arguments

all components on the FPGA and therefore reduce the net delays and increase the overall performance.

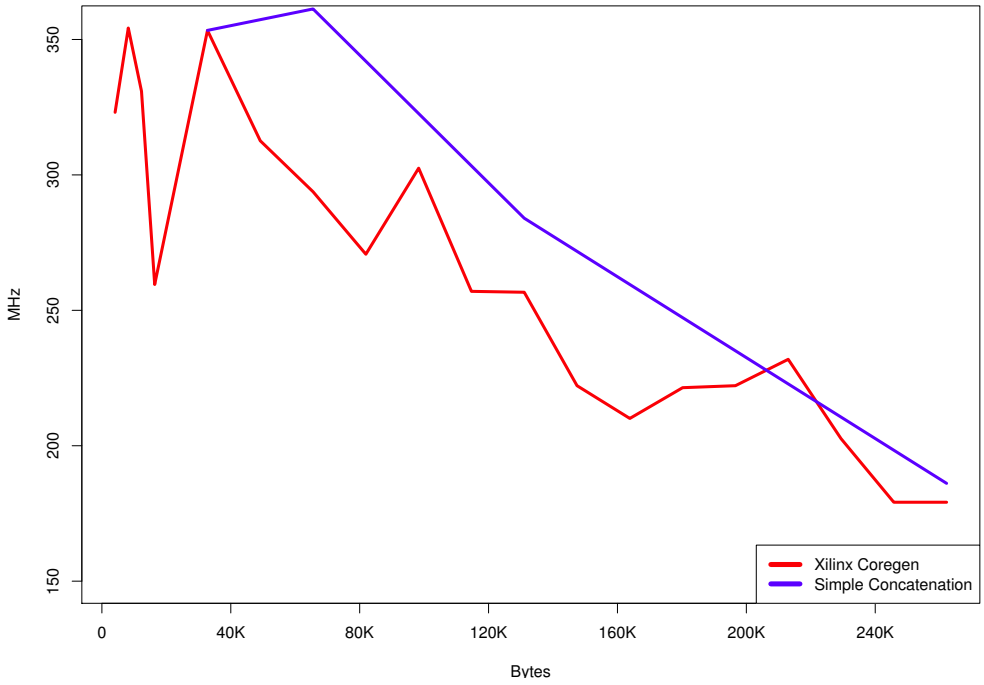


Figure 3.6.: Speed of a design utilizing different sized BRAM

The Xilinx ISE tools provide different ways to instantiate the BRAM. The Xilinx Block Memory Generator [Inc11b] can automatically create memory blocks with different parameters and the corresponding logic. We also evaluated a simple address logic that can concatenate two smaller memory blocks to a larger one. Figure 3.6 shows that a single large memory block can result in a design that is not able to achieve a clockrate of 200 MHz on a Virtex6 XC6VLX75T-3ff484. Also the core generator does not create an design that can be used with the same clock rates as the one with simple concatenation of memory blocks. A 512 kib memory block can run at a speed of 360 MHz using our own address logic, but a memory block with 1 Mib can only be used at a maximum clockrate of 280 MHz.

3.4. Executing Large Schedules using a small amount of Memory

Since the resources available to a collective offload unit are anticipated to be much smaller than the resources available to a host CPU centric collective implementation we have to deal with the problem of how to execute schedules which are small enough to fit into the host memory, but too big for the memory available on an offload unit. If this is not done it is unlikely that users would adopt offloading techniques that severely limit the size of executable collective functions, compared to traditional approaches.

First we will show that the previously suggested approach of introducing dummy operations at which larger schedules can be splitted has limits not stated in the original paper. We will describe an algorithm to detect possible deadlocks in a schedule that does not fit in the execution window of an interpreter unit. One solution to the problem of insufficient memory is to exchange synchronizing point to point transfers with RDMA operations where possible. We will describe how to detect situations where this can be done.

3.4.1. Limits of Previously Suggested Approaches

The GOAL Paper [HSL09c] states in Theorem 4:

> The space requirements to execute a schedule can be reduced to $\mathcal{O}(1)$ if dummy actions are added.
>
> Proof: Inserted dummy actions, which are not executed (i.e., they represent NOPs), can be used to introduce additional dependencies, such that:
>
> - all actions between two consecutive dummy actions i and $j (i < j)$ depend on the completion of i, and
> - dummy action j depends on all actions between i and j.
>
> Such a transformed schedule needs to remember at most $j - i - 1$ action items (equidistant dummies). The order of actions must be a valid order according to the topological sort of the original graph. All spare dependencies crossing the dummy actions can now be removed safely while

> retaining (restricting) the original dependencies. Thus, it is possible to limit any window-based scheme to have a maximum number of unreachable actions during execution.

In the following we show that this Theorem is not true in the general case. The problem with this proof is that the precondition *The order of actions must be a valid order according to the topological sort of the original graph.* is too weak. The cited paragraph clearly talks about process local GOAL graphs, otherwise the whole constant-memory bound would not make sense, as the globally aggregated memory has to be (at least) $\Omega(p)$ where p is the number of ranks that contribute to the collective expressed with the graph. We will show that process local graphs do not contain all necessary information to ensure the absence of deadlocks.

We will show examples of valid GOAL schedules that will terminate in finite time when executed with the scheduling algorithm proposed in Listing 1 of [HSL09c]. However, as soon as the above transformation is applied to those schedules, they will no longer terminate under all circumstances.

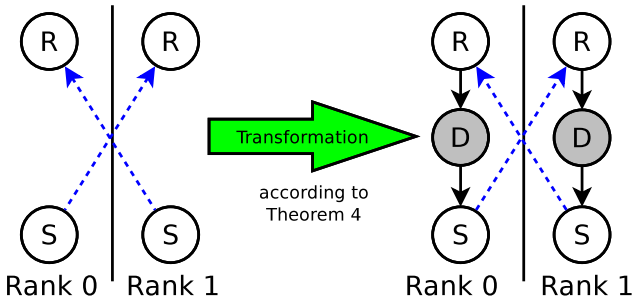


Figure 3.7.: A Simple Counterexample that cannot be transformed with the proposed algorithm

Consider the example schedules for a process group of two ranks, show in Figure 3.7. Each rank posts a send to and a receive from the other rank. There are no dependencies between the operations on each rank. The unmodified schedule is shown in the left part of the image. Note that this schedule is not only a valid GOAL schedule, it is also a very practical example for a two-process barrier. Since the cited proof states that *the order of actions must be a valid order according to the topological sort of the original graph* it would be valid to put the receives before the sends in the topological order, as they have no dependencies between them in the original graph. If we assume that our memory constraints only allow exactly one operation to be in the "execution window" at any given time (or $i - j - 1 = 1$ in the cited text), the transformed graph with added dummy operations and dependencies between dummy operations and original operations would look like the one shown in the right half of Figure 3.7. This

schedule will never terminate, as each rank will wait for data from the other rank. This cannot happen with the original graph if the scheduler is allowed to process all operations with no incoming dependencies in parallel. Note that the question which and how elements can be executed in parallel by the scheduler is an important factor. We will explain that in depth below.

Let us first consider another example where the transformations described above lead to non-terminating schedules. Under the impression of Figure 3.7, one could think that prioritizing sends will solve the deadlock problem described before. However, this approach will also fail for some schedules, for example the one shown in Figure 3.8 rank zero has to execute two receives. Those two receives will be executed in parallel in the original schedule because there is no dependency between them.

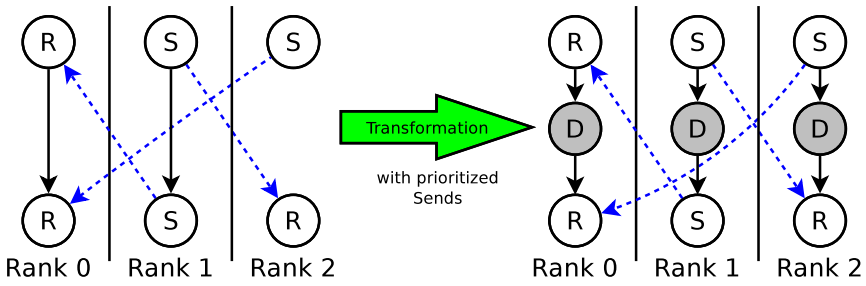


Figure 3.8.: Example schedule which cannot be transformed with the proposed algorithm if sends are blocking and synchronizing

In Figure 3.8 however, the sends on rank 1 and 2 both try to send data to ranks that have not posted the matching receive yet. Since there is an outgoing dependency on those sends they could block forever if the receive operation is implemented in a synchronizing fashion, for example because the data to be transferred is too large to be stored in local buffers.

In conclusion, the main problem with the "$\mathcal{O}(1)$-memory-theorem", Theorem 4 in [HSL09c], is that it assumes that if a schedule is valid, dependencies can be introduced between arbitrary nodes in topological order. This would be true if we would insist that every valid GOAL graph has to terminate for all topological orderings, even if all operations are performed in a synchronous way and no operations are executed in parallel. However, those strict requirements are not necessarily the way in which GOAL is supposed to work. Even the simplest schedules would be invalid for a GOAL scheduler which imposes such strong restrictions. For example it would be impossible to implement a local copy operation if the NISA only provides send and receive operations.

3.4.2. Testing for Deadlocks in Schedules

We have shown in Section 3.4 that there are schedules which may deadlock when synchronizing transfers are used. It is possible to detect such a characteristic when the global graph constructed using the local schedules of all ranks is available. It is not necessary to store the graph at a single place to start the deadlock detection [CMH83]. All dependencies in a schedule are also part of the new dependency graph, but transfers have to be converted from directed edges to dependencies in both direction to represent the synchronizing behavior of an operation. A synchronous send S and receive R have to be replaced using two new operations. A synchronous send will be splitted into a receive R_1 and a send S_2 which depends on the receive R_1. All incoming edges of S must be moved to R_1 and all outgoing edges must start at S_2. The corresponding receive R has to be splitted into a send S_1 and a receive R_2 which depends on S_1. All incoming edges of R must be moved to S_1 and all outgoing edges must start at R_2. Extra edges are inserted from S_1 to R_1 and S_2 to R_2. This mapping will ensure that the synchronizing behavior is well expressed in the dependency graph.

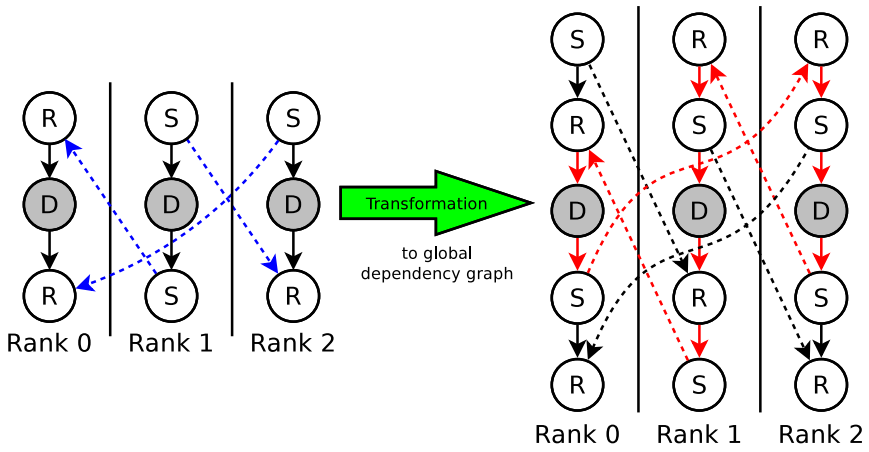


Figure 3.9.: Transformed schedule from Figure 3.8 to the global dependency graph a with the cycle marked in red

The schedule in Figure 3.8 can be transformed using the explained rule and a cycle will be detected when adding a dummy node on rank 2 and corresponding dependencies as shown in Figure 3.9. Multiple cycles can be found, but showing that at least one exist is enough to show that the schedule cannot terminate (with the current placement of dummy operations) when synchronizing sends must be used. If the size of the cycle is bigger than the size of the execution window this implies that it is impossible to reorder the dummy operations to remove the deadlock.

This detection scheme can be used to avoid deadlocks. It always has to be ensured that any cycle found in the above step is contained in one global execution window to prevent the deadlock introduced by the dummy operation. The size of the execution window is not only determined by the memory available to store the schedule, also the amount of buffers in the protocol unit are an important factor. The size of the execution window is defined as the number of operations that can be executed in parallel in a non-blocking manner. Note that the detection scheme described above cannot be used to prove the absence of deadlocks, as it is based on a single matching. However, it is possible to construct GOAL schedules with non-deterministic matching.

3.4.3. Transforming Process Local Schedules into Global Schedules

In GOAL each process specifies its local part of the global communication schedule. However, in the previous section we analyzed the global GOAL graph which includes additional edges to represent data flow or matching. We will now discuss an algorithm to construct the matching set M from the process-local sets R_i, S_i, and D_i for $0 \leq i < p$. The algorithm needs to follow local and remote (send/recv) dependencies to ensure message matches in correct order. The algorithm starts to match send and receive operations that have no dependencies. Once a match is established, the operations and their dependencies are removed from the lists. New operations that have now no dependencies can be matched. The pseudo-code is shown in Algorithm 3.

An example for matching is shown in Figure 3.10 and 3.11. Figure 3.10 shows the initial state in the first iteration. All shaded tuples are in FQ waiting to be activated.

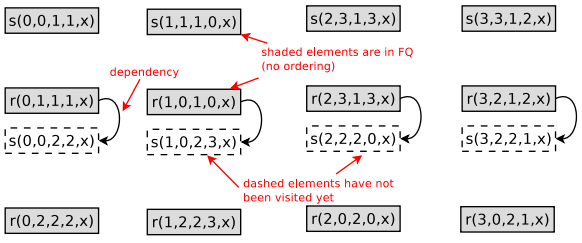


Figure 3.10.: Matching Example, initialization

Algorithm 3: Matching Algorithm

Input: List of process-local GOAL graphs G_i, one for each process as: a set of receives (R_i), a set of sends (S_i), and a set of dependencies (D_i).

Output: Set of matches (M).

```
1  FQ ← all independent nodes //unsorted list of nodes;
2  ∀i : AQ_i ← ∅ //AQ is a sorted list of nodes;
3  while FQ ≠ ∅ do
4      a ← pop(FQ);
5      y ← (ā_type, a_peer, a_owner, a_size, a_tag) //invert type, swap peer and owner;
6      if y ∈ AQ_{a_peer} then
7          AQ_{a_peer} ← AQ_{a_peer} \ y;
8          remove_deps(a, D_{a_owner}, FQ);
9          remove_deps(y, D_{a_peer}, FQ);
10         M ← M ∪ (a, y);
11     else
12         AQ_{a_owner} ← AQ_{a_owner} ∪ a;
```

Procedure `remove_deps(`*n, P, FQ*`)`

Input: Node n, set of dependencies D, list FQ

```
1  forall (n, x) ∈ D do
2      D ← D \ (n, x);
3      if ∀y : (y, x) ∉ D then FQ ← FQ ∪ x
```

Figure 3.11 shows an intermediate state of the algorithm where the first receive of process 0 is matched. After matching, the dependency is removed and the second send to process 2 is added to FQ (shaded grey).

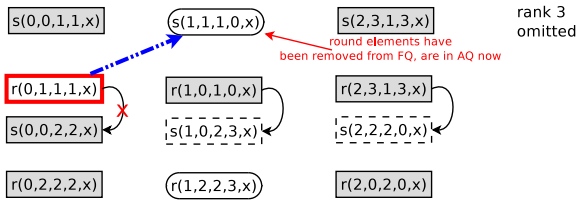


Figure 3.11.: Matching Example, after a few steps

Note that this algorithm constructs one possible matching set. It is however possible to create GOAL schedules which a non-deterministic with regard to matching. Analyzing all possible matchings is not computationally feasible because the number of possible matchings is not polynomially bounded, even for simple GOAL graphs.

We will now discuss the runtime of the Algorithm 3. Each node is inserted in FQ exactly once and in each iteration of the while loop in Line 3 one node is removed from there. Therefore the loop will be executed $\mathcal{O}(n)$ times. When a is removed from FQ in Line 4 there are two possibilities: Either we find a matching node in $AQ_{a_{peer}}$ (Lines 7–10) or not (Lines 12). The cost for the search operation in $AQ_{a_{peer}}$ (Line 6), if AQs are implemented as a red-black trees, is $\mathcal{O}(\log |AQ_{a_{peer}}|)$. Therefore we can sum up the cost for all such lookups with $\mathcal{O}(n \log n)$. This term also covers the cost for all deletions and insertions into AQs (Lines 7 and 12). In *remove_deps*, each of the $|D| < m$ dependencies is removed exactly once and each of the send/recv operations is visited once. Therefore the costs for all calls to *remove_deps*() can be summarized as $\mathcal{O}(n + m)$. The costs for the initialization (Lines 1+2) is $\mathcal{O}(n + m)$ while the summarized costs for insert and delete operations on FQ and M (Lines 4 and 14) are $\mathcal{O}(n)$, as those can be implemented as lists since no search operations on those data structures are needed. Altogether the cost for the matching algorithm is therefore $\mathcal{O}(n \log n + m)$.

3.4.4. Predetermined Buffer Locations

All scenarios that would create a global deadlock are related to synchronous send and receives. This happens due to the problem that no side of the transfer has the complete information about the source and target address. It is also possible that

only one side knows the actual transfer size. Units which track the posted receive or send operations have to be used to match the current transfers to correctly coordinate memory access. Execution would block when the operation on the remote rank cannot be found or no temporary buffer is available in case of an receive. Matching also increases the latency of a transfer.

It is possible to provide an additional operation next to send and receive which is one-sided and provides all necessary information for a transfer. Network interface cards like EXTOLL provide the necessary RDMA Get or Put functionality and therefore the remote schedule interpreter unit does not have to be contacted for such a transfer. This can remove a possible deadlock and can reduce the point-to-point latency [SBM+05].

The schedule still has to ensure that the access on remote memory is correct when operations gets executed. A remote memory operation without any synchronization may try to access a memory region which is not allocated or does not contain the correct data. Also buffers which get used multiple time may get overridden by an RDMA Put before another operation on the old data finished.

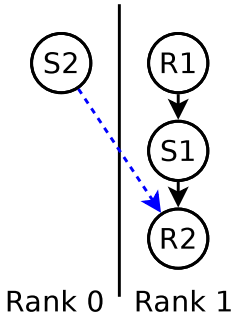


Figure 3.12.: Operations which manipulate or use a single buffer on rank 1

Figure 3.12 shows an example of two local schedules using two-sided transfers. Rank 1 uses the same buffer for all operations and therefore dependencies were added between the operations to prevent the execution of two conflicting operations at the same time. Send S_1 and Receive R_1 are two-sided operations with other ranks not shown in this example. An invalid optimization would be to determine the target address of the transfer $S_2 \rightarrow R_2$ and replacing both operations with a single RDMA Put on Rank 0. No synchronization happens between Rank 0 and Rank 1 and therefore Rank 0 cannot decide when it is safe to start the transfer. Also Rank 1 does not know when the RDMA Put is finished and its local schedule can also be finished.

Extra operations for the synchronization would have to be placed around the RDMA Put operation. A receive waiting for a start signal can be placed before the Put and a

send to inform Rank 1 about the finished operation can be placed after the Put. The receive R_2 can be replaced with a send to inform Rank 0 about the available buffer and a receive to wait for the finished transfer.

The extra synchronization steps may create more overhead than the original implementation. A schedule with multiple non-conflicting transfers between two ranks can benefit from the one-sided operation with predetermined buffer locations as long as the necessary synchronization does not create more delay. Figure 3.13 shows a schedule with a global schedule that allows to replace some operations with RDMA put operations without adding additional synchronization operations. The transfer $S_1 \rightarrow R_1$ already informs Rank 1 that the target buffers for receive R_2 and R_3 can be used. $S_4 \rightarrow R_4$ also informs Rank 0 that all transfers in this schedule are finished. This makes it possible to transform transfer $S_2 \rightarrow R_2$ to a single RDMA Put P_2 and $S_3 \rightarrow R_3$ to a RDMA Put P_3 without any additional synchronization.

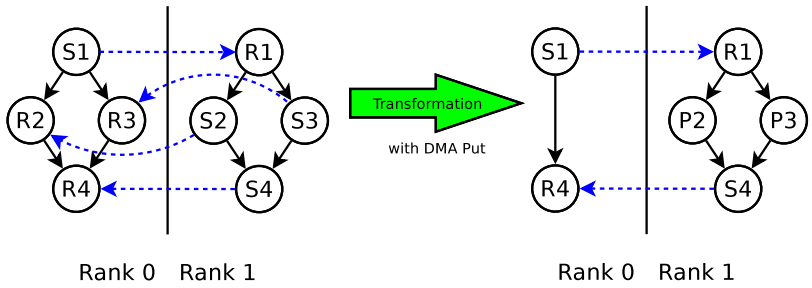


Figure 3.13.: Transformation of send-receive pairs using available synchronization

It is not necessary to consider the direction of a synchronization transfer when both send and receive operation are synchronizing. Both ranks know after a finished receive or send that the remote rank also started the transfer.

3.5. Queueing Active Operations in Hardware

The interpreter unit in hardware must read all independent operations of a schedule and try to start them using extra units. It is more than likely that the extra units cannot buffer all operations which are independent. The interpreter still has to ensure that no operations are lost during the run of the schedule. Thus the interpreter must provide queues or other kinds of memory which can be used as buffers for the operation management.

There are two different types of operation buffers which should be managed by the interpreter unit directly: preposted network operations and independent operations. Preposted network operations are operations which were already managed by the protocol unit and independent actions are operations which still have to be started by a protocol unit. Both are not hard requirements because it is possible to use only the schedule stored in BRAM with a flag to save whether the operation is already finished. Combining the dependency counter and the flag is enough to find finished operations and operations which can be started. A unit would have to go through the complete schedule to decide what must be done to continue with the schedule.

Such a strategy would have the problem that for each finished operation the whole schedule has to be parsed again to find operations which could have been started, but were not started due to the limited resources. This increases the time needed to execute the schedule in comparison to usage of dynamic datastructures for handling these queues. Those are not trivial to implement in hardware and could lead to deadlocks or invalid execution when implemented with the wrong parameters, such as not enough memory. A unit that finished an operation could not inform the unit operating on the schedule because this unit waits for the queue to become non-full again to insert new operations that can be started. A deadlock would have been created when the unit which handles the starting of the new operations is either the same as the unit which wants to insert finished operations or is also blocked by the unit which wants to insert the finished operations.

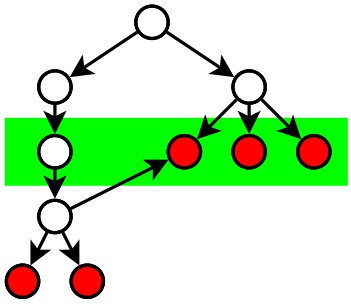


Figure 3.14.: Maximum wavefront of a breadth-first search in green and the worst case of operations to store in queue

The execution of operations in a single memory block of schedule are only ordered by dependencies between them, the order of processing of finished operations and the starting of independent operations. This could result in a execution where each finished operation only triggers the start of a single operation or were a single finished operations results in the start of all remaining operations. The upper limit of operations which must be stored is similar to the maximum wavefront in a breadth-first search, but not equal to them. Figure 3.14 shows an example directed acyclic

graph with the maximum of 4 operations in the wavefront and 5 operations for the worst case queue usage.

It is possible to calculate the maximum possible number of elements which must be stored in the queue, but this calculation would only be valid for a schedule or a partial schedule that was created for execution with $\mathcal{O}(1)$ memory as shown in Section 3.4. The schedule would need to be resplitted in case that the calculation showed that the queue must be bigger than the available memory. It is also possible to calculate the maximum number n of operations in a schedule which still fits inside the limited amount of memory together with the queues. This would ensure that we always have enough room in the queue to store all operations of a schedule as explained in Section 3.3, but may leave a lot of memory unused in case of a linear list of operations.

A queue can be implemented using a linear block of memory with one read and one write port and two registers as explained in Section 5.1.1. Both registers are set to zero on reset which represents the relative position inside the memory block. When an element is added using the write port then the register which represents the tail pointer is incremented and the new element is written at that position. The head register points to the element which will be read next and is incremented after each read operation. The queue will not create an overflow after a reset, for example because a new schedule block is started.

3.6. Designing a Low-Memory-Footprint Point to Point Protocol

If we offload the execution of GOAL schedules to specialized hardware units we have to ensure that the amount of memory needed to store unexpected messages and control information about messages which are currently being transferred is small. Latency is another important factor when choosing a point to point protocol: In many situations it is possible to use less buffer space if we ensure synchronization between the sender and the receiver. This additional synchronization can only be achieved by means of sending additional messages, which in turn increases the point to point latency.

MPI implementations have to solve a similar problem as we do when deciding how to implement MPI_Isend() and MPI_Irecv() for networks (like EXTOLL) which have RDMA support [LWP04, SJCP06, Pak08]. But there are also some important

differences that forbid us to simply use the same protocol as any RDMA-aware MPI implementation: MPI implementations are typically executed on a host CPU. Therefore unexpected messages can easily be stored in the host memory. For our implementation this would lead to several problems. We cannot guarantee that all ranks which participate in a collective will start the execution of that collective simultaneously. It is likely that, for example due to process skew, the root node of a collective is still executing an old schedule, while all other ranks already loaded the next one. If this new schedule contains a send to the root rank in the independent action list this would result in $p - 1$ messages arriving at the root which cannot be matched yet. Therefore they have to be stored somewhere. One possible solution would be to allocate a huge buffer on the host memory during initialization and store all unexpected messages in that buffer. This solution looks simple at first but has serious implications: Matching those unexpected messages would also have to be done by the host CPU later, as the GOAL interpreter does not have the memory resources to keep a copy of each message header. Therefore our initial goal of offloading as much as possible of the collective execution would be compromised.

Also the progression rules are different, while it is acceptable for an MPI implementation to rely on polling to finish a message transfer, i.e., copy a message from a temporary to the destination buffer, the communication protocol utilized for GOAL cannot do that, as all polling would have to be done by the protocol state machine itself.

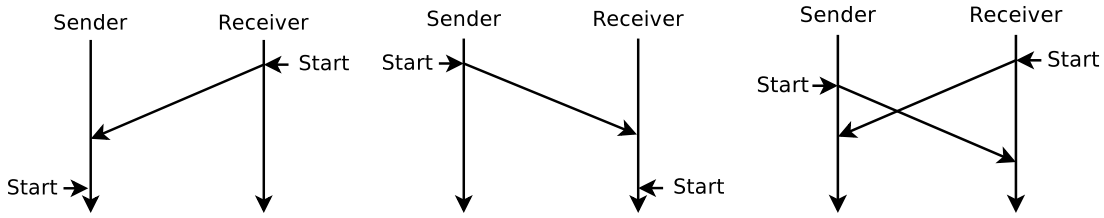
In the remainder of this section we will describe a new point to point protocol which does not need to store any information about unexpected messages. The general idea of that protocol is that if a rank wants to send data to another host he signals his intent by a small control message which includes all necessary information needed by the receiver to decide if he can store the data in its destination buffer or not (because the receiver has not loaded the schedule in question yet or the corresponding receive has not been posted). If the receiver cannot accept the data he can just ignore this packet. Once the receiver becomes ready he will signal this with a small message to the sender. Only if the sender is ready to send and the receiver is ready to receive data will be transmitted. The receiver is in control of the transfer, he will fetch the data using a RDMA_Get() operation.

To enable the reader to understand this protocol we will first introduce some terminology and introduce two commonly used point to point protocols. We will describe why both protocols are sub optimal for our GOAL offload unit and describe our new protocol. To ensure that our protocol works in the way we expected we used protocol verification techniques. We will explain the working principles of the tools used for this task and describe some of the problems we discovered during the verification phase. Finally we will describe the architecture of the point to point protocol which we

use for the hardware GOAL interpreter unit. The actual implementation is described in detail in Section 5.3.

3.6.1. Arrival Times

For the discussion of protocols we differentiate between three different scenarios. The scenarios are defined by the start time of the communication task of the sender and receiver, relative to each other. The different scenarios are depicted in Figure 3.15.



(a) Late Sender (b) Late Receiver (c) Similar Arrival Times

Figure 3.15.: Different relative arrival time scenarios

The sender is regarded to be "late" (Figure 3.15(a)) if it is possible for the receiver to send a control message (i.e., a 0B message) to the sender, and the message arrives there before the sender enters the point to point communication. In the LogP model the arrival times for this scenario satisfy the equation $t_{sender} \geq t_{receiver} + L + 2o$.

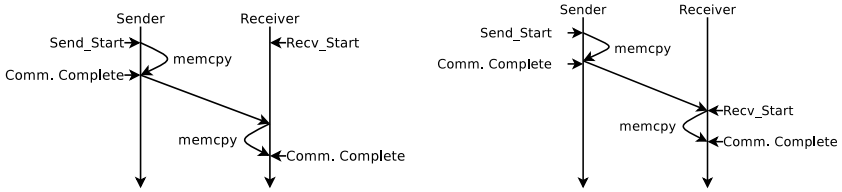
Similarly, if the arrival times satisfy the equation $t_{receiver} \geq t_{sender} + L + 2o$, as shown in Figure 3.15(b), the receiver is regarded as "late".

If neither of the above conditions hold $|t_{sender} - t_{receiver}| < L + 2o$ must be true. This scenario is shown in Figure 3.15(c).

3.6.2. Eager Protocol

The *eager* protocol is often used for the transfer of small messages. For those it is unacceptable to pay more than a single network latency for each transmission. Therefore no synchronization step can be done before the message is transmitted. As a result the sender does not know the address of the destination buffer. Instead of transferring the data to the (unknown) destination, temporary buffers are used. For fastest completion on both sides the sender copies the message to a temporary buffer

and attaches the necessary header, such as the length of the data. It then sends it to a temporary buffer on the receiver side. Upon reception the message has to be copied from the temporary buffer to the destination buffer, once it is known. Two examples for this are given in Figure 3.16.



(a) Simultaneous arrival of sender and receiver

(b) Late Receiver

Figure 3.16.: Examples for the eager protocol

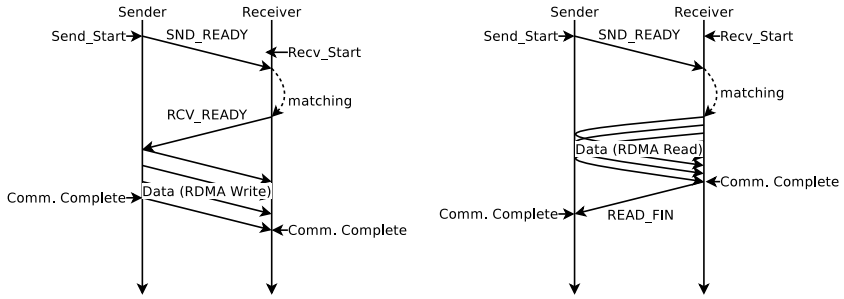
This protocol is very fast. On the sender side the time needed for the transmission is only the time required to perform the local copying of the message to the temporary buffer. This time can be summarized under o_s. On the receiving side the time taken is at most $L + 2o_s$. The general disadvantage of this protocol is that we need to allocate a lot of temporary buffer space on each node and it incurs a high CPU overhead to do the local memory copy after the matching of each message is done.

With this protocol every received message has to be matched against the *preposted-receives* queue. If a matching receive is found in that queue the message can be copied to its destination buffer. If no matching receive is found, the message is copied to a *unexpected-messages* queue. If a receive is posted, the unexpected-messages-queue has to be searched for that message first, to check if the message has already been received.

In the context of our hardware GOAL interpreter the unexpected-messages queue is problematic because we do not have enough memory resources in hardware to store the incoming messages.

3.6.3. Rendezvous Protocol

Rendezvous protocols do not require temporary buffer space for message data as rendezvous protocols ensure that the receive is already posted before the send is started [shi06]. This implies that neither the sender nor the receiver can finish their part of the transaction before the other side starts the operation.



(a) RDMA-Write based rendezvous protocol

(b) RDMA-Read based rendezvous protocol

Figure 3.17.: Examples for the rendezvous protocol

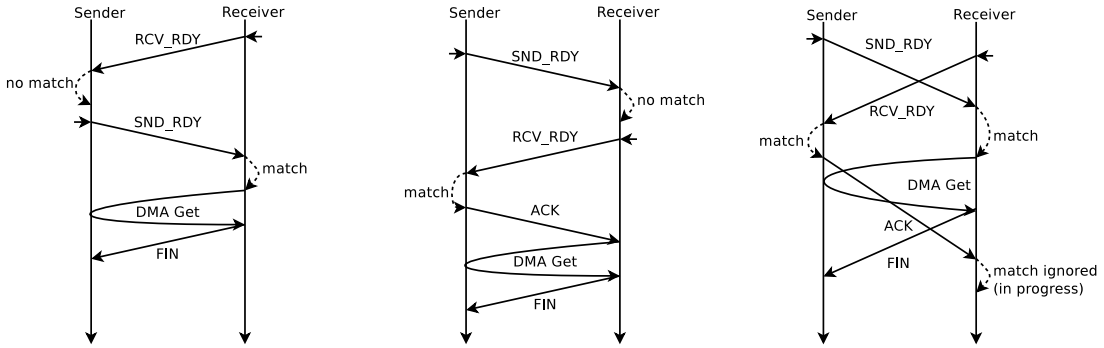
If RDMA-Write is used to transfer the data the sender takes control over the transfer and knows when he can reuse the send buffer, since he can track when the last byte of data was sent. This is shown in Figure 3.17(a). In the case where RDMA-Read is used for the transfer, the sender does not have this information as the receiver controls the transfer, therefore the receiver has to notify the sender with a special control message that the send buffer may be reused now. Note that an RDMA implementation, such as the RMA-Unit [NSB09] in EXTOLL provides low level hardware support for such notifications.

The problem with this protocol is that, similar to the eager protocol described in Section 3.6.2, we need two queues: The unexpected message queue and the preposted receive queue. The preposted receive queue can never contain more entries than the number of receive operations in the currently active schedule(s). The unexpected queue however can contain as many entries as there are active sends in all nodes together. Of course in a (correct) schedule we will eventually have a receive for each unexpected message but it is possible that other ranks start a schedule before the local rank does, for example because the local rank does not have enough resources — however the resulting unexpected messages cannot be dropped if this protocol is in use.

3.6.4. A Protocol without an Unexpected Queue

Since the management of the arbitrarily large unexpected queue that is needed for the simple RDMA based point to point messaging protocol explained above poses numerous problems, we designed a new protocol which limits the size of the required queues by imposing stricter synchronization semantics. In particular we remove the

unexpected queue. All unexpected messages are dropped and both sides signal their readiness to the communication peer by means of an additional control message. The protocol is depicted in Figure 3.18 for the three different arrival time scenarios.



(a) Receiver early (b) Sender early (c) Simultaneous

Figure 3.18.: An RDMA based point to point protocol without the need for an unexpected message queue

The working principles of that protocol are described below. Note that the protocol is slightly more complicated in practice — the basic protocol is not sufficient to guarantee deadlock freedom in schedules. We found and fixed those problems by means of protocol verification. Our protocol verification techniques are described in Section 3.7.

Sender side protocol The sender transmits the header and data location of the send operation, as soon as it is started (SND_RDY packet) and enqueues that header in a *posted-sends* queue.

The sender always listens to RCV_RDY packets which contain the header that is expected by the receiver. If such a packet is received the header is matched against the posted-sends queue. If a match occurs the sender signals that the data is available with an ACK packet. Otherwise the message is ignored.

Note that each packet can be marked with a flag chosen by the originator of the message which is copied in the reply to facilitate fast "matching" of request-response messages.

Receiver side protocol As soon as the receiver starts the communication it sends a RCV_RDY message with the expected message header. This header is enqueued in the *posted-recvs* queue.

The receiver always listens to SND_RDY messages. If a SND_RDY message is received it is matched against the posted-recvs queue. If a match occurs the data is fetched with a DMA put operation and a notification (FIN message) is sent to the sender side. The receiver has to ensure to keep track of finished messages to ensure that the data is not fetched twice in the case where sender and receiver arrive simultaneously.

3.7. Protocol Verification

To ensure that the newly designed protocol works as intended under all anticipated circumstances we decided to use formal verification. Another approach suggested to us was to implement the proposed protocol in a high level language such as C and see if it is still working as intended after a large number of transmissions. However we are convinced that such an approach will hardly find all corner cases of a formal protocol specification.

There is a large number of tools which are designed to provide conclusive proves about properties of models. Some of the well known ones are SPIN [Hol97], Nu-VMS [CCG+02] and VIS [BHSV+96]. We choose to use the model checker SPIN, based on the large user base in the scientific community. In the following we will describe how SPIN works and describe the first and simplest SPIN model of our protocol. After that we will describe the problems that we detected due to the use of formal verification techniques and also explain how those problems were eliminated in the actual protocol implementation.

3.7.1. Capabilities of the Model Checker SPIN

A SPIN model consists of one or more processes. Each process is a nondeterministic finite state machine. The verification process works by running all state machines together, until all reachable global states have occurred once. The user can state invariants that should always remain true during the verification process. If SPIN detects a global state where a specified invariant does not hold it aborts the verification

and tries to find the shortest list of state transitions that lead to the invalid global state.

To give an example of the capabilities of SPIN we will examine the mutual exclusion algorithm published in [Hym66]. The algorithm is shown in Listing 5.

Algorithm 5: Incorrect mutual exclusion algorithm

Input: Process id i either 0 or 1

```
1  b_i ← 0, ∀i ∈ 0, 1;
2  j ← 1 − i;
3  k ← 0;
4  while 1 do
5      b_i ← 0;
6      while k ≠ i do
7          while b_j = 0 do
8          k ← i;
9      critical section;
10     b_i ← 1;
11     remainder of program;
```

To verify (or disprove) the correctness of Algorithm 5 we have to transform it into a representation that SPIN can understand. The input language of the SPIN model checker is called PROMELA. Since PROMELA models will be used to generate non-deterministic finite automata there is an important difference in PROMELAs syntax compared to imperative programming languages such as C: Non-determinism can be expressed by statements beginning with `::`. All of those statements in a basic block can be executed. However, there can also be a condition for the executability of a statement which is expressed in PROMELA in the form of `:: condition -> statement`. Therefore a PROMELA fragment that generates a random number between zero and four could look like Listing 3.1.

Listing 3.1: Example for generating a random number in PROMELA

```
1 int a = 0;
  do
3   :: a = (a + 1) % 5;
    :: break;
5 od;
```

In each iteration of the do-loop the NFA has the "choice" to increment the variable a again or to leave the loop. Note that during verification such a fragment will lead to 5 different global states, as a can assume any integer value between zero and four.

A PROMELA model of Algorithm 5 is given in Listing 3.2. When the critical section is entered we increment a global counter which holds the number of processes which are currently in the critical section. After that we check if the counters value is equal to one. If SPIN can find a path to a global state where this assertion is violated the mutual exclusion algorithm does not work, since both processes are in the critical section at the same time. The execution of a process in SPIN is non blocking of course, so that the init process launches two instances of PROCESS simultaneously.

Listing 3.2: A PROMELA model of Algorithm 5

```
byte b[2];
int k = 0;
int critical = 0;

proctype PROCESS (bit i) {
  int j;
  j=1-i;
  do
    :: b[i] = 0;
    do
      :: k == i -> break;
      :: else ->
        do
          :: b[j] != 0 -> break;
          :: else -> skip;
        od;
        k = i;
    od;
    critical++; assert(critical == 1); critical--;
  b[i] = 1;
  od;
}

init {
  b[0] = 0; b[1] = 0;
  run PROCESS(0); run PROCESS(1);
}
```

From that Listing SPIN will generate the NFA shown in Figure 3.19. Note that this NFA is actually a deterministic finite automaton (DFA), which is not surprising, given that the algorithm was intended to work on deterministic computers and does not involve (pseudo-)randomness.

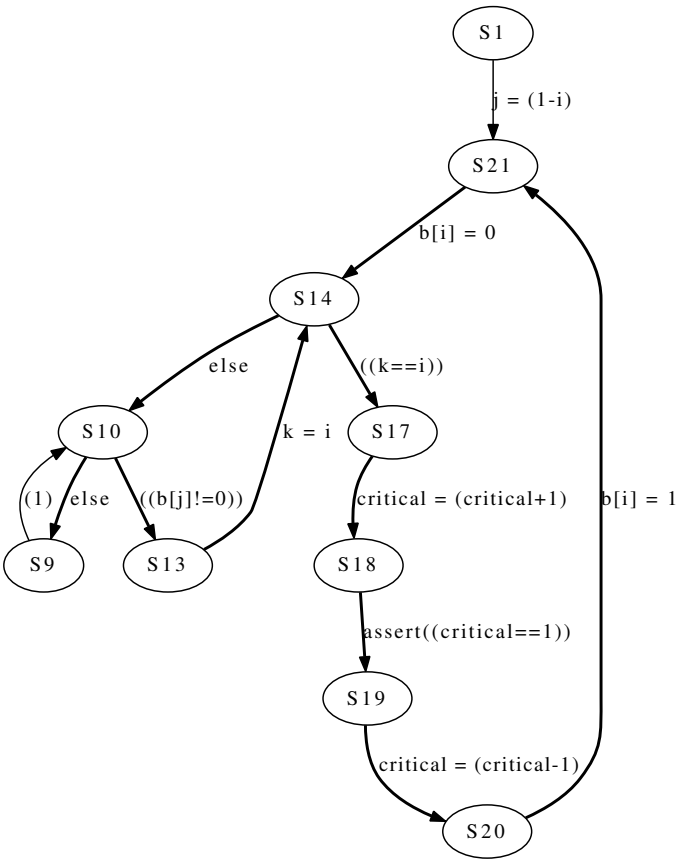


Figure 3.19.: The NFA generated by SPIN from Listing 5 for verification

SPIN is able to verify the behavior of this model in a fraction of a second. As expected the assertion about the number of processes in the critical section is violated. It is also possible to create an error trail for that assertion violation, for which SPIN will try to find the shortest execution path that leads to the violation. For the given example an excerpt of the error trail is shown below. For readability we marked the execution trace of the processes 1 and 2 in different colors.

```
1 Starting PROCESS with pid 1
  proc 1 (PROCESS) (state 1) [j = (1-i)]
3 Starting PROCESS with pid 2
  proc 2 (PROCESS) (state 1) [j = (1-i)]
5 proc 2 (PROCESS) (state 2) [b[i] = 0]
  proc 2 (PROCESS) (state 5) [else]
7 proc 1 (PROCESS) (state 2) [b[i] = 0]
  proc 1 (PROCESS) (state 3) [((k==i))]
9 proc 1 (PROCESS) (state 17) [critical = (critical+1)]
```

```
   proc 1 (PROCESS) (state 18) [assert((critical==1))]
11 proc 1 (PROCESS) (state 19) [critical = (critical-1)]
   proc 1 (PROCESS) (state 20) [b[i] = 1]
13 proc 2 (PROCESS) (state 6) [((b[j]!=0))]
   proc 1 (PROCESS) (state 2) [b[i] = 0]
15 proc 1 (PROCESS) (state 3) [((k==i))]
   proc 2 (PROCESS) (state 13) [k = i]
17 proc 2 (PROCESS) (state 3) [((k==i))]
   proc 2 (PROCESS) (state 17) [critical = (critical+1)]
19 proc 1 (PROCESS) (state 17) [critical = (critical+1)]
   spin: test.pml, Error: assertion violated
21 spin: text of failed assertion: assert((critical==1))
```

From the trace we can see that the main problem of Algorithm 5 is that the checks $k \neq i$ and $b_j = 0$ are not executed in an atomic context. So it can happen that a process gets interrupted by the other process between these two checks. The analysis of error trails was an important step in designing our point to point protocol. It enabled us to iteratively refine and correct our model.

3.7.2. Modeling the Protocol

To verify the protocol proposed in Section 3.6.4 different models with increasing complexity were implemented in Promela. These models were able to initiate multiple send and receive pairs. The models were designed as follows: There are two distinct GOAL units, each connected to a model of the "transceiver interface", the COMM unit. All communication between GOAL units is done through those transceivers. In our model we connected the two COMM units directly to each other, a network was not modeled. A overview over the connected units is given in Figure 3.20.

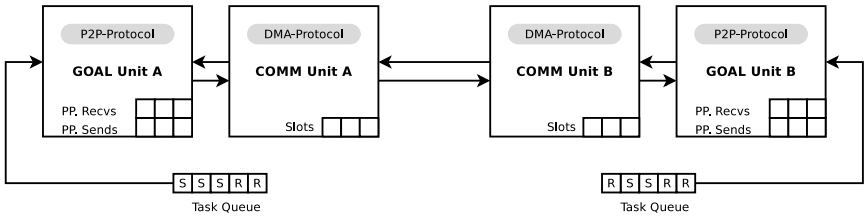


Figure 3.20.: Overview of the functional units involved in the verification model for our new point to point protocol

Each GOAL unit has a queue of tasks attached to it. This queue specifies which network primitives (send or receive) should be started in which order. The model implementation initializes both protocols in such a way that each send has a matching receive on the peer side and vice versa. The ratio of sends to receive operations as well as the matching for each operation is chosen by the user. The order of the network operations in the queues is randomized before the execution of the model. The amount of buffers for transfers which are in progress (named *slots* in the following) are configurable.

The simplified NFA for the protocol as executed by the GOAL unit is shown in Figure 3.21.

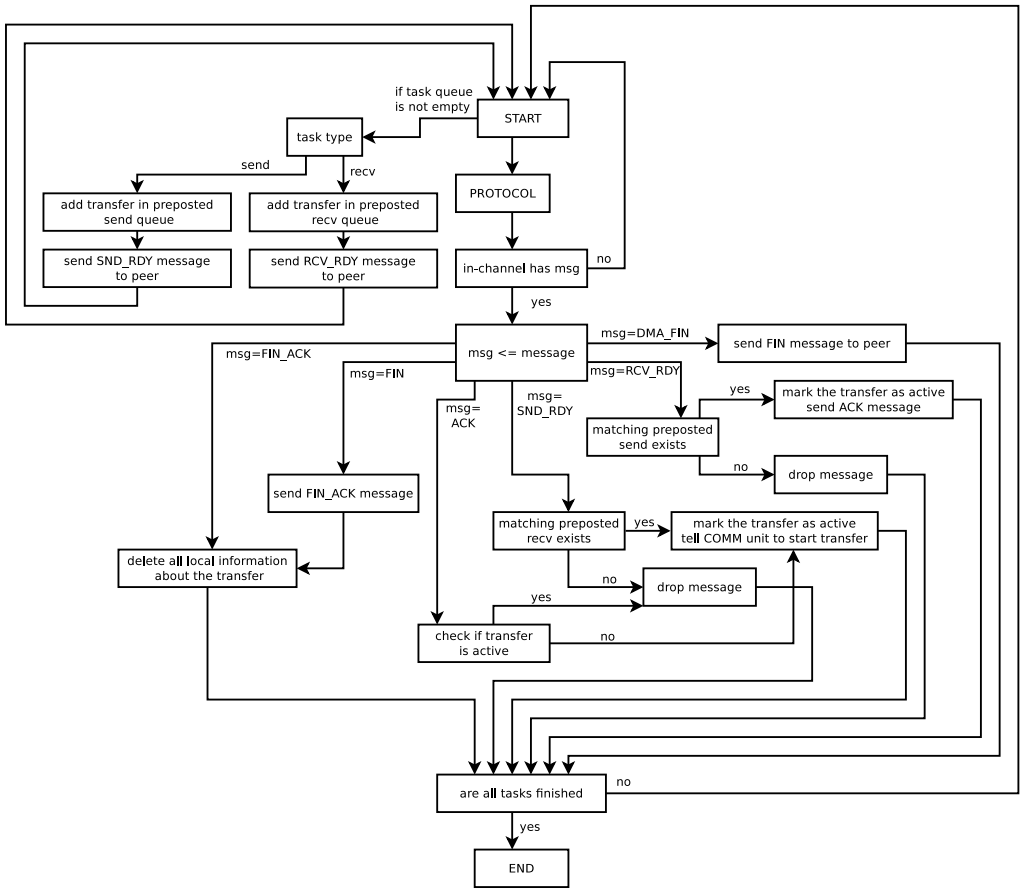


Figure 3.21.: NFA for the verification of the GOAL units point to point protocol

The NFA for the COMM unit is not shown, in the verification model we assume that the COLL unit is capable of buffering the same amount of in-progress transfers as the GOAL unit. The COLL unit verification model also simulates the completion

of transfers: A transfer can complete at any time after it was started, this implies that transfers do not necessarily complete in the same order in which they have been started. Note that the NFA handles messages (FIN, FIN_ACK) which are not explicitly mentioned in Section 3.6.4. A rationale for the necessity of these messages is given in below.

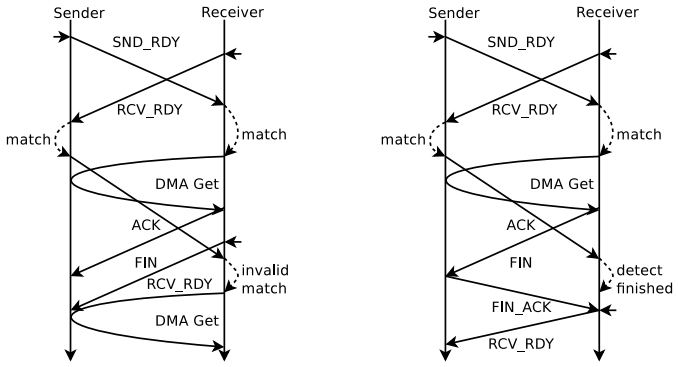
3.7.3. Limitations of the Basic Protocol

The analysis of the protocol showed that the information about already finished messages is needed to ensure that a send is only matched once. It is also important that the entry in the preposted queue can only be deleted when all messages from the sender side associated with this transfer were received. That includes a acknowledgment that the sender is also informed about the receiver side state. We propose different enhancement and clarifications to the protocol to solve the problem of pairwise not distinct matching elements processed by the matching units. For correctness it is required that the receiver side never starts transfers when it gets wrong information from the sender side and no transfer stops when temporarily inconsistent states exist.

Tracking already finished transfers

The basic protocol included the requirement that already finished transfers must be tracked until no side will ever again operate on that information. It never answered were such information is tracked and how it can be decided that there do not exist conflicting information about the state of a transfer. Figure 3.22(a) shows an example were it is possible that a RCV_RDY triggers an ACK, but the ACK is received after the transfer was already finished and the next transfer was started. This would result in a too early start of the next transfer when we assume that the information about the already finished transfer was dropped.

The protocol can use the order property of command messages to add another packet type called FIN_ACK which informs the receiver that the sender side has sent all packets related to this transfer, knows that it was finished and no new information related to this transfer will be sent to the receiver side. The arrival of this packet is the first safe state where the receiver side can also drop any information about this finished transfer. This slot can now be reused by another transfer without accidentally matching it with an outdated in-flight packet.



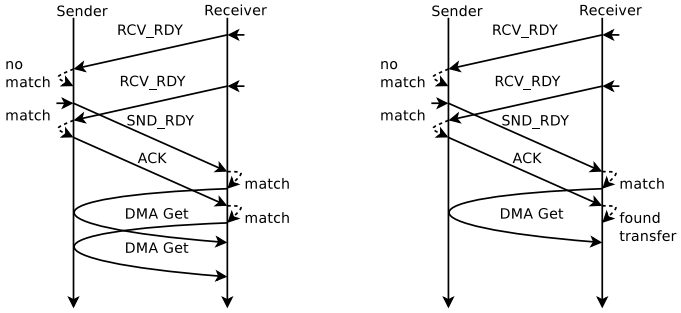
(a) Invalid matching of ACK with a new started transfer

(b) Freeing of finished transfers after synchronization with sender side

Figure 3.22.: Late arrival of ACK packets with early removal of finished transfers

Double Matching Sends

A similar problem arises when two or more receives with the same matching element are started and the sender side starts the corresponding send after the first RCV_RDY was dropped, but before the second RCV_RDY is processed. Figure 3.23(a) shows that after the SND_RDY also a ACK is send and the receiver side can now match it against two different transfers and start two different RDMA Gets.



(a) Matching of ACKs by receiver slotid

(b) Finding conflicting slots when receiving ACKs

Figure 3.23.: Matching of a single send with two different receives

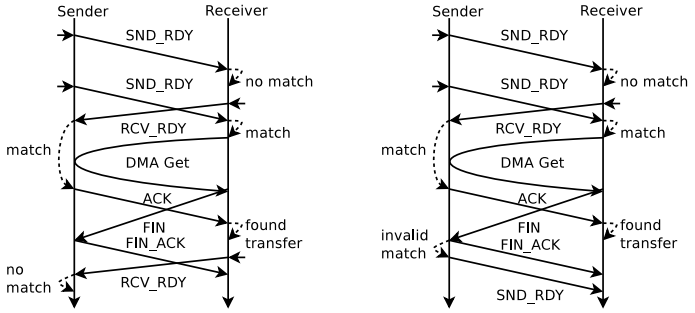
The solution is to search for slots in the preposted receive queue with the same sender slot when receiving ACKs or SND_RDY packets as shown in Figure 3.23(b). Those conflicting information have to be dropped to prevent the additional RDMA Get.

The check for SND_RDY is not obvious when assuming that the SND_RDY is the first packet the receiver side gets when the sender initiate a send and no command messages are reordered. It is still needed for a protocol change explained in the following sections.

Sender-side only matched transfer

The dropping of acknowledgments now creates the problem that necessary acknowledgments are lost. Figure 3.24(a) shows an example were the sender transmits two SND_RDY and the receiver starts matching RCV_RDY for both after the first SND_RDY was dropped, but before the second SND_RDY was processed.

The second SND_RDY will be matched to the only receive in the preposted receive queue, but the sender will assume that this RCV_RDY matches his first entry in his preposted send queue belonging to the first SND_RDY. This inconsistent state will result in an dropped acknowledgment when the receiver matches the sender slot id with already active or finished transfers in his preposted receive queue.



(a) Acknowledgment dropped due to inconsistent matching

(b) Finding conflicting slots when receiving FINs

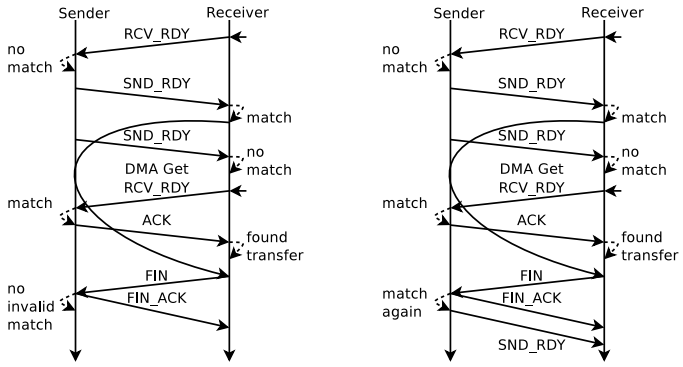
Figure 3.24.: Invalid matched transfers on sender-side

The infinitely waiting preposted send can only be prevented when information about the possible matching preposted receive queue slot is stored on the sender side. This information has to be saved when the sender found a match after a RCV_RDY. Only a FIN packet will trigger a check if the information about the receiver slot id in the received command matches the saved receive queue slot id in the preposted send queue. Figure 3.24(b) shows that the actual finished slot had a different receiver slot id stored and all other transfers with the receive slot id from the FIN packet were restarted.

Sender-side missed match

The search for invalid matches does not solve the problem of invalid information on the sender side. Figure 3.25(a) shows an example were the missing information about the started transfer by the receiver can result in a completely unmatched transfer on both sides.

The receiver has to send his RCV_RDY before the sender has started the corresponding transfer. This information will be dropped, but the receiver can automatically match the SND_RDY with his preposted receive without informing the sender about it. The next SND_RDY command is now again without a matching preposted receive on the receiver side, but the later posted RCV_RDY can match against the first preposted send because the sender does not know anything the ongoing receiver controlled transfer. The acknowledgment check on the receiver side will detect this problem and drop the ACK packet. Now it is not possible anymore that the receiver will send the corresponding RCV_RDY and the sender has no information that the second preposted receive will never be matched.



(a) Missing match on sender-side due to false positive match

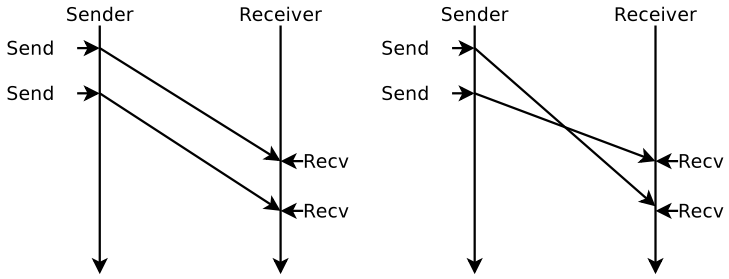
(b) Finding slots with same matching element when receiving FINs

Figure 3.25.: Missing matched transfer on sender-side

The problem can be solved when the sender extends the consistency check on incoming FIN packets so that it does not only look for other entries with the same receiver slot id, but also for preposted sends which were not yet matched and have the same matching element as the now finished transfer. The detected possible stalled transfers can be restarted as shown in Figure 3.25(b). Information about transfers which already have been started by the receiver side will automatically be dropped by the receiver-side without any further interaction.

4. The Matching Problem

The matching problem has to be solved by all message passing frameworks. Matching in this context means to determine which receive operation should finish upon arrival data. Messaging frameworks such as MPI or GOAL define a set of rules, which have to be followed for matching. One common matching rule is that messages can only match in their own process group and that messages have to be matched in the same order in which they were posted. Note that GOAL does not require strict ordering, it only has to be ensured that the matching does not violate the user defined dependencies between operations. See Figure 4.1 for an explanation: Without the matching rule on ordering it would be possible for messages to overtake each other in the network. Commonly the user can influence the matching using tags. For example the matching shown in Figure 4.1(b) could be forced by the user by giving both messages a distinct tag.



(a) Ordered matching: A message sent first is also matched first

(b) Out-of-order matching: Messages can overtake each other

Figure 4.1.: Impact of different matching rules on the applications data flow pattern

To implement matching rules message passing frameworks put all posted receive operations in a *preposted receives queue*. Upon the arrival of a message this queue is traversed the new message is tied to the first receive found in the queue where all matching conditions (such as source rank and tag) are met. After that (and before the next incoming message is matched) the receive is removed from the preposted queue to ensure a bijective mapping between receive and send operations. The matching process is performance critical for a message passing framework, especially since most implementations do not utilize a protocol as the one described in Section 3.6.4 which

does not need an unexpected message queue and therefore not only have to enqueue posted receives but also messages that could not be matched upon arrival. If the application programmer is not careful this unexpected message queue can become large. To speed up the matching process "matching acceleration units" have been proposed, for example in [UHR+05]. A problem that arises when offloading the matching process is that the memory available on an FPGA or matching ASIC is almost certainly smaller than to host memory. This problem could be solved by using the host memory as "swap space" for queue elements that do not fit into the hardware matcher. Such an approach was proposed in [UHR+05]. This approach poses a new problem: To make good use of the available memory bandwidth to the host memory queue parts should be swapped in and out in contiguous blocks of memory. However, since new entries into the matching queues always have to be inserted at the end of the queue but entries are deleted at arbitrary positions such contiguous blocks could easily degenerate into blocks holding almost no valid entries after a number of insert and delete operations.

For this reason we developed the point to point protocol described in Section 3.6.4 which does not need an unexpected queue. The preposted receive queue is limited by the size of the schedule, as it has to hold at most one entry for each receive operation in the schedule. In the following we will benchmark the time needed for the processing of the matching queues on the host CPU. We will then describe the Verilog implementation of a matching unit.

4.1. Matching on the Host CPU

If we want to implement a matching unit in hardware we need a way to compare its performance to the traditional approach of performing matching on the host CPU. For this purpose we analyzed the matching process in Open MPI version 1.4.3. Open MPI uses the Modular Component Architecture (MCA) design principle. An overview over some of the components that make up Open MPI is given in Figure 4.2.

The semantics of MPI_Send() and MPI_Recv() are provided by the Point-to-point Management Layer (pml) component. Therefore this layer also has to match remote sends to local receives and vice versa. However there are network interface cards and driver stacks that already provide a matching interface in their low level API, for example Myrinet or Portals. Therefore Open MPI offers multiple pml modules, the ob1 pml for example is the default pml module and does not offer any special features but also does not impose requirements on the byte transport layer (btl) in Open MPI

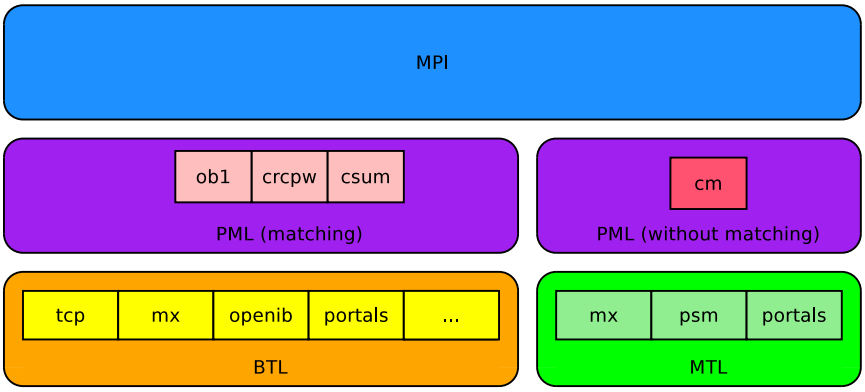


Figure 4.2.: Overview of the Open MPI Architecture

terms. The cm pml however does not perform message matching as it relies on the matching transport layer (mtl) to do that.

There are other pml components which are all forks of the ob1 pml and provide additional features such as support for coordinated checkpoint-restart (crcpw pml), verifying message integrity at the pml layer via checksumming (csum pml), or providing fault tolerance by logging and replaying messages (v pml).

The ob1 pml component provides the matching MPI Send/Recv interface in the following manner: If a new message is received it checks if the destination communicator was already created. If not, the message is stored in the non_existing_communicator_pending queue of the pml. If the communicator already exists the incoming message is matched against the receives already posted by the target rank in the target communicator. If no match is found the message is placed in the unexpected_frags queue at the target. If a new receive is posted the pml searches the unexpected queue for a matching message and places the receive in the preposted receive queue if no match was found. The non_existing_communicator_pending queue is searched when a new communicator is created, messages which were addressed to the newly created communicator are copied to the unexpected_frags queue of the destination rank.

In most cases the actual matching process only requires few comparisons: Two messages match if they either have the same tag or one has the tag MPI_ANY_TAG and the other one has a tag value greater than zero. Negative tag values are used internally to implement collectives. This ensures that point to point messages belonging to collectives cannot match user specified receives and vice versa. The check if the communicators and ranks are equal does not have to be done since the ob1 pml uses a separate queue for each MPI process.

It is clear that queue operations are an integral part of an MPI implementation. Open MPI implements all mentioned queues as double linked lists. The list implementation is done in OPAL, the Open Portability Access Layer, which implements different datastructures in an object oriented fashion. To get an impression of the performance of these queue walking operations we copied the preposted receive queue implementation, also using the same data type for the stored data elements, and wrote a small benchmark code around it.

Our benchmark works in the following way: First the queue is filled with n elements. After the initialization the timer is started an we will traverse the entire queue. After that loop is completed the timer is stopped. We are capable of performing cycle accurate timing by utilizing the hrtimer library also used in netgauge [HMLR07]. Each round (with a different value for n) is run 100 times and the median of the obtained results is reported. The results are shown in Figure 4.3.

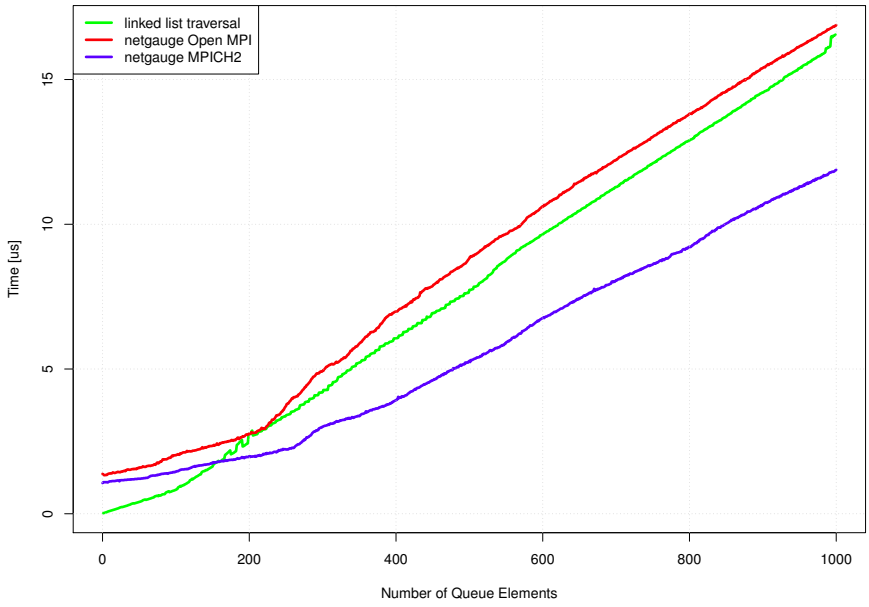


Figure 4.3.: Performance of message matching on a Host CPU core

This benchmark method has the obvious drawback that it benchmarks the performance of the message matching process as it is implemented in one MPI implementation. And it is cumbersome and error prone to isolate the message matching code in an MPI implementation. For example in our case we overlooked the fact that Open MPI uses its own memory allocation functions instead of simply calling malloc() to create new list elements. The positive aspect about the benchmark method described above is that there are no other codepieces that contribute to the measurement result.

However, to be able to compare different MPI implementations it would be useful to have a message matching benchmark which is independent of the actual MPI implementation, relying only on behavior that is specified in the MPI standard. Such a benchmark can be constructed as shown in Figure 4.4 and has been implemented in the netgauge benchmark tool. The benchmark works as follows:

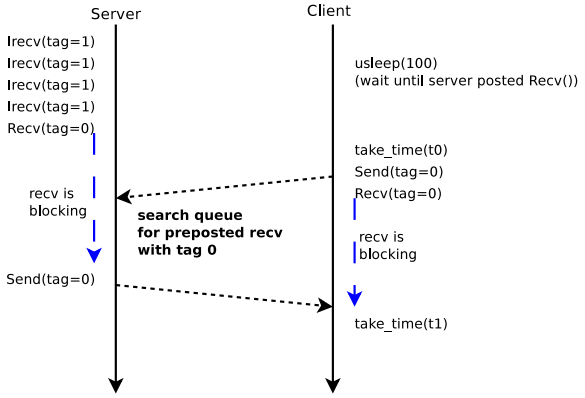


Figure 4.4.: Message matching benchmark in netgauge

For each number of elements n, the "server" rank posts n MPI_Irecv(), all with a tag value of one. After that the server rank is doing a MPI_Recv(), immediately followed by an MPI_Send(), both with tag zero. The client part will sleep for some time to make sure that the server has already arrived at the MPI_Recv(). After that he is taking a timestamp and starts a MPI_Send() to the server, followed by an MPI_Recv(), both with tag 0. After that another timestamp is taken. To match the incoming message, the server has to traverse the matching queue to find the earliest matching receive. Only after that the call to MPI_Recv() can return on the server side and the MPI_Send() can be executed.

So the time measured by the client is the time for the matching process, the time for one ping-pong roundtrip plus any extra overhead incurred by the messaging. If we fit linear models of the form $t \sim \alpha n + \beta$ to our measurement data we get the following results:

Benchmark	α [μs **per Element**]
Open MPI (list_test)	0.016995
Open MPI (ng)	0.017013
MPICH2 (ng)	0.011680

All benchmarks described above have been carried out on a dual CPU, dual core AMD Opteron 285 CHiC [MMHR07] cluster node with 4 GB of RAM. We used Open MPI 1.4.2 and MPICH2 1.2.7 as packaged in Debian squeeze.

If we want to achieve a similar matching time per element (0.01 μs) for our hardware unit and assume that the hardware unit runs at 200 MHz the hardware unit can spend at most two clock cycles per queue element comparison.

4.2. Implementation Methodology

When implementing the hardware GOAL interpreter we had two important requirements: The Verilog design has to be synthesizeable and we need precise control over the timing behavior of the resulting design. The first requirement stems from the fact that only synthesizeable designs can be evaluated with regard to the clock speed achievable with that design on a particular hardware platform. Verilog contains several constructs which are, albeit useful, not synthesizeable. Examples are *initial blocks* or *wait* statements. Verilog also offers some constructs which can be synthesized but are not directly mapped to logic gates but rather cause the synthesis tool to instantiate sequential logic with a certain behavior. Examples for such constructs are loops or certain forms of if statements. If such statements are used extensively it is hard to keep track of the registers that were inferred by the synthesis tool. It is possible that the synthesis tool infers latches while the developers goal was to implement only combinatorial logic. It is hard to check if the synthesis resulted in a circuit that actually has the desired properties. It can be done by viewing the synthesis result in the Xilinx *FPGA Editor* or in the *PlanAhead* tool (cf. 2.7). However, for larger designs this process is cumbersome and error prone. It can be compared to hand-optimizing a numerical computation by tweaking C code and viewing the compilers output in a disassembler.

To avoid those problems we chose to adopt a design pattern called *Algorithmic State Machine Design*. First we tried to break up larger functional units into small modules and sketched each modules tasks. Then we designed the datapath for each module. While designing the datapath it is important to keep an eye on the path length between registers: A large number of logic gates between registers will introduce logic and routing delay along that path, therefore buffer registers were introduced on problematic paths. For example paths from larger memory blocks are the performance bottleneck for most of our modules as address decoding logic will be introduced by the synthesis tool along that path. Each modules behavior is governed by a single finite state machine, called *control unit* in the following text. The control unit is implemented as a finite state machine. There are several different ways to implement finite state machines in a hardware description language [Gol94] — the choice which design pattern should be used depends on the abilities of the synthesis tool as well

as on personal preference. We used a state machine design pattern called *Mealy Automaton*. The basic structure of such a state machine is depicted in Figure 4.5.

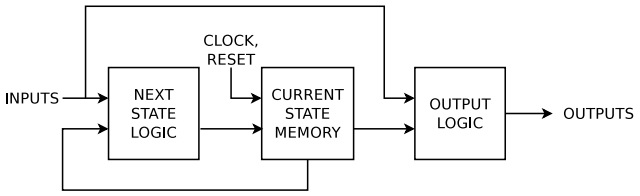


Figure 4.5.: Structure of a Mealy Automaton

Note that the only part in such an automaton that uses flip flops is the current state memory. An example for the Verilog implementation of a very small mealy automaton is given in Listing 4.1. The block that starts with *assign next_state* represents the next state logic. We ensure that this block only infers combinatorial logic by using the Verilog keyword assign and the ternary operator ? instead of using an if ... else or switch statement. The next_state is saved in the register state which holds the current state in the always block. The always block is triggered by a positive clock edge on the clock signal. Furthermore the always block handles the reset. The output logic is defined by assign blocks such as the one starting with *assign busy*. Again, this block should only contain combinatorial paths. Note that the values that define each state are not hardcoded in this implementation but defined as parameters (comparable to a define statement in C code). The Xilinx ISE tool can optimize the state encoding, for example the synthesis tool might be determine that for this state machine a one-hot state encoding would be more efficient and will re-encode the states before synthesis [Inc09].

Listing 4.1: Simple FSM implemented in Verilog

```
1  module my_automaton(
     output wire busy,
3    input wire start,
     input wire [1:0] din,
5    input wire clk,
     input wire reset
7      );

9    wire [2:0] next_state;
     reg [2:0] state;
11
     parameter STATE_0 = 3'd0;
13   parameter STATE_1 = 3'd1;
     ...
15   parameter STATE_X = 3'dx;
```

```
17   assign busy =
       (state == STATE_0) ? 1'd0 :
19     (state == STATE_1) ? 1'd1 :
       ...
21     ((state == STATE_2) && (din == 2'd1)) ? 1'd1 :
       1'dx;
23
   assign next_state =
25     ((state == STATE_0) && (start != 1)) ? STATE_0 :
       ((state == STATE_0) && (start == 1)) ? STATE_1 :
27       ...
       STATE_X;
29
   always @(posedge clk) begin
31     if (reset == 1) begin
         state = STATE_0;
33     end
       else begin
35       state = next_state;
       end
37   end
endmodule
```

It is obvious that such an FSM implementation is both hard to read and write. Therefore we developed a tool which is capable of reading a table in text form, where each cell contains the logic function for a signal (defined by the row) when in a certain state (defined by the column). When given such a table as input the script will generate the equivalent assign statements for each output signal. During the design phase however, it is most helpful to work with a visual representation of the statemachine instead of a tabular one.

4.3. Matching Unit Interface

The point to point messaging protocol implementation described in Section 5.3 needs a functional unit that is capable of managing the protocol state of several in-flight message transfers. We call each saved state of such an in-flight message a *slot*. Slot managing and message matching are closely related. For example the point to point protocol needs the possibility to check if a slot with the given peer id and remote slot id exists. This is necessary to detect whether the RDMA transfer was already

started as explained in Section 3.7.3. For message matching we need the possibility to check if a slot with the specified communicator, schedule, tag and peer rank exists. Therefore we decided to implement message matching and "slot matching" in a single unit, called matching unit in the following text.

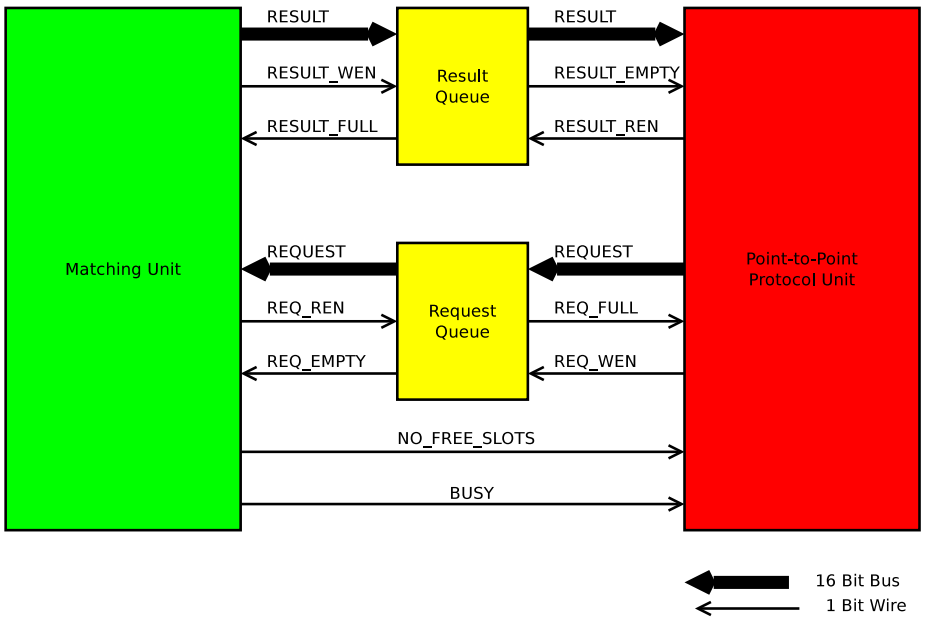
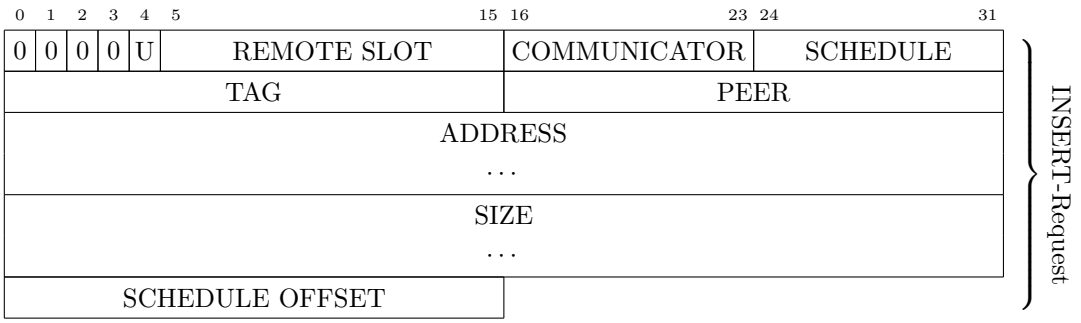


Figure 4.6.: Interface offered by the matching unit

The interface to the matching unit is packet based. That means that we define a request packet for each feature offered by the matching unit. The interface of the matching unit is shown schematically in Figure 4.6. Requests are sent over the 16 Bit request bus. Responses are emitted over the 16 Bit response bus. To decouple the matching unit from the protocol unit small queues are used to buffer request and response packets. The matching unit supports 9 different functions. Each function corresponds to a specific request packet. Those packets are described below.

To add a new slot to the matching unit, the following packet has to be sent:



The response to such a packet is the slot id of the newly created slot. This id is also called local slot id.

If the new slot is added in a situation where not all values are known yet, the corresponding bits in the command packet can be filled with any value, however, the field may not be omitted. The internal status of the newly created slot will be set to INACTIVE.

To delete a slot the following packet has to be used:

This will immediately free one slot in the matching unit if the specified local slot id was valid. If the given local slot id was invalid or the matching unit is busy the request is ignored. This packet generates no response.

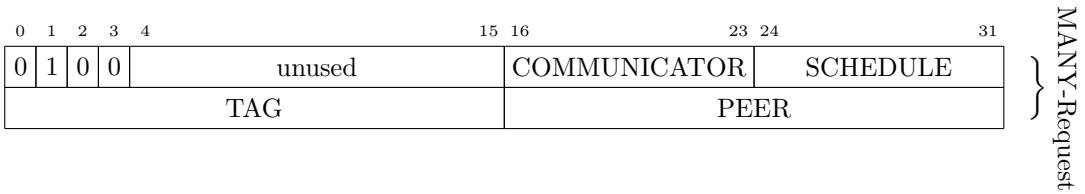
The status of a slot (ACTIVE or INACTIVE) can be queried with the following command:

If the slot with the given local slot has the status ACTIVE the following response packet is created:

If the slot was in the status INACTIVE no response will be generated. Newly inserted slots are marked as inactive by the matching unit until their status is set automatically to ACTIVE or using the activate-packet:

This command does not generate a response in any case. If the matching unit is not busy and the local slot id is valid the state of the slot will be set to active.

To find a slot which matches a posted receive or send operation the match_any request has to be used:



This request will cause the matching unit to search through all slots and return the local slot id of the first slot which has the status INACTIVE and where the comm, sched, tag and peer fields are equal to the values specified. The status of the slot with the local slot id returned will be set to ACTIVE by the matching unit. If no match was found no response will be generated. The response packets generated by the match_any command have the following form:

The point to point messaging protocol also needs to check if an ACTIVE slot with a given remote slot and peer identifiers exists. This can be done with the following packet:

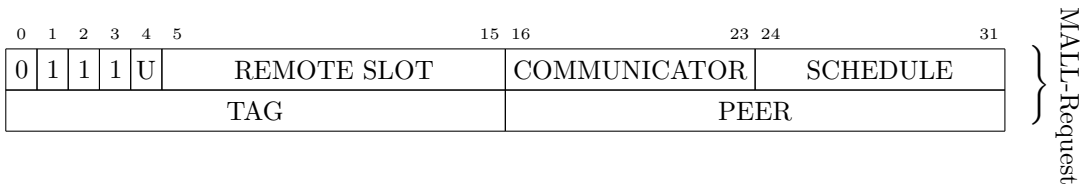
If a matching slot was found a response packet of the form shown below is created:

Otherwise this command generates no response.

Furthermore the ability to change the remote slot id is needed, because this information is typically unavailable when the slot created. For that purpose the following packet has to be used:

This packet will change the remote slot id of the slot identified by the given local slot id. This packet does not trigger a response.

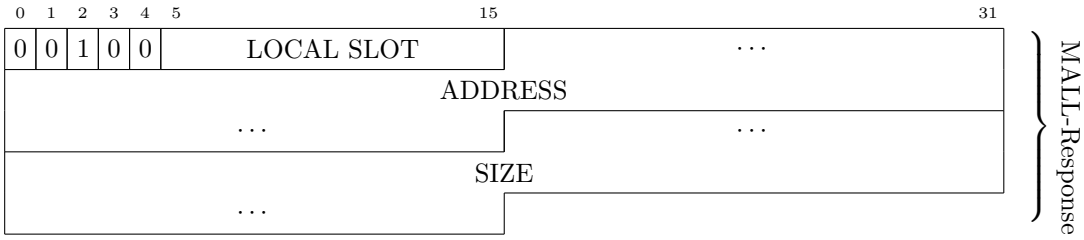
In cases of "false matching" as explained in Section 3.7.3, the point to point protocol needs to find all slots that are possibly affected by that matching. These are queried with the match_all command packet:



This packet will search for slots s where either

$$
\begin{aligned}
s_{comm} &= \text{COMMUNICATOR} \wedge \\
s_{sched} &= \text{SCHEDULE} \wedge \\
s_{peer} &= \text{PEER} \wedge \\
s_{tag} &= \text{TAG} \wedge \\
s_{status} &= \text{INACTIVE} \\
&\text{— or —} \\
s_{peer} &= \text{PEER} \wedge \\
s_{remote\_slot} &= \text{REMOTE SLOT} \wedge \\
s_{status} &= \text{ACTIVE}
\end{aligned}
$$

is satisfied. All matches will be returned consecutively. If no match occurred this command generates no answer. The status of each matching slot is set to INACTIVE. The response packets for this command have the following form:



The point to point protocol also needs read access to most fields in the slots. For example, address and size fields most be read when the matching unit found a valid entry after receiving a SND_RDY or RCV_RDY packet. Also a FIN or FIN_ACK packet would result in the read of the schedule offset which is not transferred as part of the protocol packet. Therefore the matching unit implements a read field command:

Where FIELD ID is defined as follows:

FIELD ID	Meaning	Response Length
0x1	Schedule+Commit ID	16 bit
0x2	Tag	16 bit
0x3	Peer	16 bit
0x4	Remote Slot	11 bit
0x5	Schedule Offset	16 bit
0x6	Buffer Address	64 bit
0x7	Buffer Size	64 bit
0x8	Remote Slot, Address, Size	139 bit
0x9-0xF	unused	—

4.4. Matching Unit Implementation

From the observations made in section 4.1 it is clear that the matching unit should be capable of comparing an entry in the matching queue each clock cycle to offer a similar performance than host based matching. Since the matching element is 64 bit wide we need the possibility to test for a match of over 64 bit in a single cycle. The queue elements are therefore stored in a memory unit which has a 64 bit wide data bus. This means that for insertion of new queue elements we need another functional unit to receive the insertion request (which is given in 16 Bit packets) and write the data into the *slot memory* once a 64 Bit chunk of data is ready. We call this functional unit the *input consumer*.

Figure 4.7 gives an overview over the different components of the matching unit. Several commands implemented by the matching unit result in a response. This

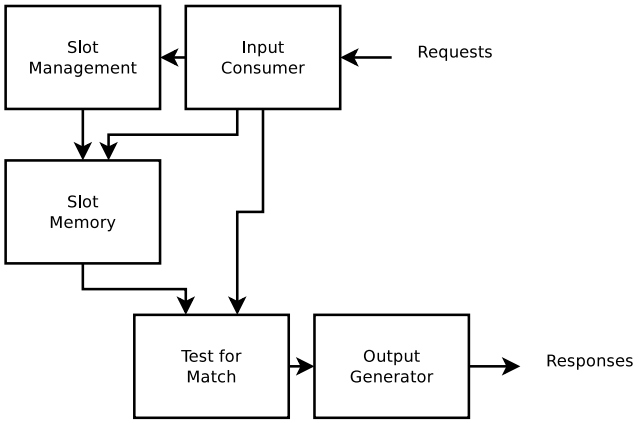


Figure 4.7.: Overview of the Matching Units components

response is generated by a functional unit called *output generator*. An important component of the matcher is the *slot management unit*. It has to perform several different tasks: When a new slot is inserted with the insert command, it will provide the address where the new slot should be inserted. Upon deletion it will mark the address of the deleted slot as free again. During matching the slot management component will iterate over all addresses that are occupied. so that the slots can be compared with the matching element given in the match-all or match-any request, which is stored in the input consumer. We will describe the slot management unit in detail below.

4.4.1. Slot Management Unit

The slot management unit has to support the following operation:

- Return an address of a free slot and mark it as allocated
- Mark a given allocated slot as free
- Iterate over all allocated slots, in the same order in which they have been allocated
- Check if there are free slots available

To the best of our knowledge, there is no datastructure that is capable of performing all operations in constant time. The usage of datastructures which use indirections,

such as a double linked list is also problematic since such indirections would lower the throughput of the iterate operation. Therefore we chose to implement the list management unit based on two queues: the *free list* and the *used list*. The idea is that the free list contains all addresses of empty slots, while the used list contains pointers to all used slots. If a slot is allocated the first entry in the free list is removed and inserted into the used list. For iterating we just iterate over the contents of the used list. Freeing a slot is a more complicated operation, as it involves iterating over the content of the used list until the right slot is found, then this address is removed from the used list and inserted into the free list.

Free List

We use fixed-size blocks to store slot data. The task of the free list unit is to keep track of unused blocks. Since slots are inserted and deleted in no particular order we use a queue to store all pointers to free memory blocks. The free list has to support four different operations:

The *allocate* operation returns the address of any unused memory block. It may only be invoked if such a block is available. After a block has been allocated it cannot be allocated again before it is freed by the *free* operation. This operation marks the memory block which starts at the given address as free. This operation may only be used on memory blocks that have been allocated before. The *empty* operation checks if there are free blocks available in the free list. The free list has to be initialized by invoking the *reset* operation. The reset operation can be invoked any time. After a reset all memory blocks are regarded as free.

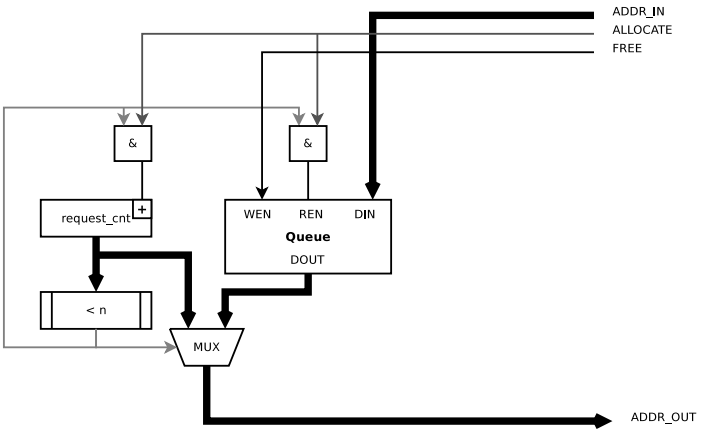


Figure 4.8.: Block Diagram of the Free List Management Unit

Such a unit can easily be implemented as a stack or queue, where the allocate operation pops an element, free adds an element and reset pushes all memory block addresses. The problem with this approach is that the runtime of the reset operation becomes linear in the number of available addresses. This problem can be solved using the invariant that an element that has never been popped before is empty. Therefore we can do the following:

Suppose we have to manage n memory blocks. Then the first n allocate request will return the addresses $0 \dots n-1$. If a free request occurs it is always pushed on the stack. After the n'th allocate all requests are serviced by popping and pushing from the stack. Using this approach the reset operation only has to clear the stack pointer and the request counter. The allocate operation has to check the request counter to decide if an element has to be popped from the stack or if the next free block address is determined by the value of the request counter. However, in hardware this decision can be done using only combinatorial logic. A schematic overview of the free list is given in Figure 4.8.

Instead of a stack, a double ended queue is used in our hardware implementation. This has the advantage that we have two data ports instead of one, so the logic can be simplified. The queue design is the same as for the Active Queue described in Section 5.1.1.

Used List

As described above we use a free list to manage empty slot space. This gives us the ability to quickly find the next available slot. For the matching process we need the opposite: we have to find the next used slot. This is done with the Used List. Its task is to store addresses of used slots in the slot memory. If a slot is freed, the entire list has to be traversed and the item that corresponds to the freed element has to be removed. Therefore the used list can not be implemented in combinatorial logic, it needs a state machine for the queue traversal. The state machine is shown in Figure 4.10. A diagram showing the data flow of the unit is given in Figure 4.9. Obviously the control unit containing the state machine shown in Figure 4.10 is not shown in the dataflow overview for simplicity's sake.

If an allocate operation is performed (the allocate input wire is set to one) the address that should be allocated is given at the data in (din) bus. Therefore mux0 has to route din to the din input of the queue in state zero. This operation takes one clock cycle. The free operation works by assigning the address of the slot that should be deallocated at the din port and assigning the free signal to one for one clock cycle.

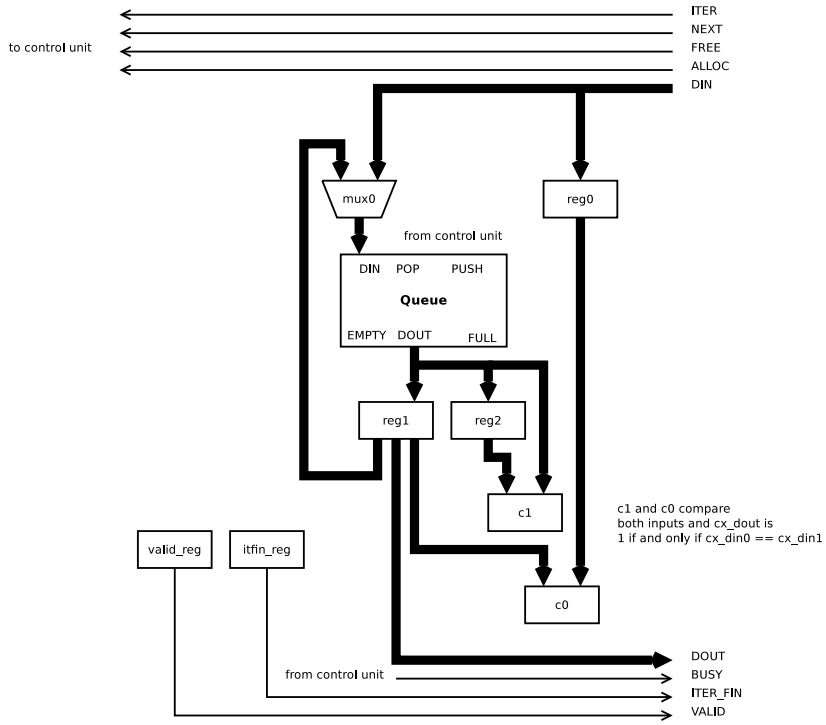


Figure 4.9.: Data Flow of the Used List Management Unit

The address is then saved in the reg0 register. In the next cycle (state one) the topmost element is popped from the queue and stored in the reg1 register in state two (the queue has a read latency of one cycle). In state three the comperator c0 checks if the element that should be removed from the queue was found, if yes we return to state zero, if not, the address temporarily stored in reg1 is pushed back on the double ended queue in state four and execution is continued at state one by popping the next element from the queue. For the matching process we need the ability to iterate over the contents of the used list and reading each address from the data out (dout) bus. However, in most cases we do not want to iterate over all entries, after a match is found it is unnecessary to iterate any further. Furthermore we need to know if the iterate process is finished because each element was already iterated over once. This is done with the input signals the first is done with the input signals iterate, next and abort_iterate. The end of an iterate loop is signaled via the output iter_fin. If the unit which instantiates the used list wants to start an iteration loop it has to set the iter input signal to one for one clock cycle. This causes the statemachine to pop the first element from the queue if such an element exists, i.e., if the queue is currently not empty. The validity of the values at dout is signaled via the valid output wire. Note that valid is set to one for only one clock cycle. This ensures that we do not generate a double match if the value at the dout bus is not updated in

the next clock cycle. If the instantiating unit wants to read the next value during an iteration loop it has to signal this by setting the next input to one. If an iterate loop is started the queue head is placed in the register reg2 in state five. During iteration the current element is always stored in reg1 and compared to the first element in reg2 via the comperator c1. If the output of c1 becomes one during the iteration loop, the contents of reg1 are pushed back into the queue and the end of the iteration loop is indicated in state six. Note that during the iteration loop, which happens in state 7, a new queue element is assigned to dout in each clock cycle. This gives us the ability to do check for a match of a queue element in each clock cycle. The statemachine that defines the behavior of the used list is shown in Figure 4.10.

In this Figure, dashed rectangles contain assignments that are done with combinatorial logic (dependent on the current state). The things in solid rectangles happen at the state change (positive clock edge) and can be thought of as comments on what is happening in that state. Note that we omitted values of signals that are not important for a state from the state diagram. The next-state function is represented with arrows between the different states. If a state transition is conditional the conditions are written at the arrows between the combinatorial block of the state and its clock edge blocks. In addition the next state is indicated in the upper left corner of each clock edge block. In the next section we will describe how the free list and used list are used together to form the slot management unit.

Slot Management Unit

The slot management unit utilizes the free list and the used list to manage the available slots. It has to ensure that both lists which hold complimentary data stay synchronized — if an entry is added to the free list it has to be removed from the used list and vice versa.

From the dataflow diagram in Figure 4.11 it can be seen that the data out (dout) ports of both lists are connected to the slotreg register. If a new slot is allocated a free address is requested from the free list in state zero of the slotmanagers statemachine (shown in Figure 4.12). This address is stored in the slotreg register in state one. In state two the slot is inserted into the used list. Therefore it is ensured that the lists are synchronized again after an allocation. If the slot management unit is requested to free a certain slot, its address is written into the slotreg register in state zero. In state four both lists are notified that the corresponding slot should be marked as free. The free list can do that immediately, since it only has to append an entry to its internal queue. The used list may need more time for this operation, therefore the slot management unit loops in state five until the busy signal emitted by the used list

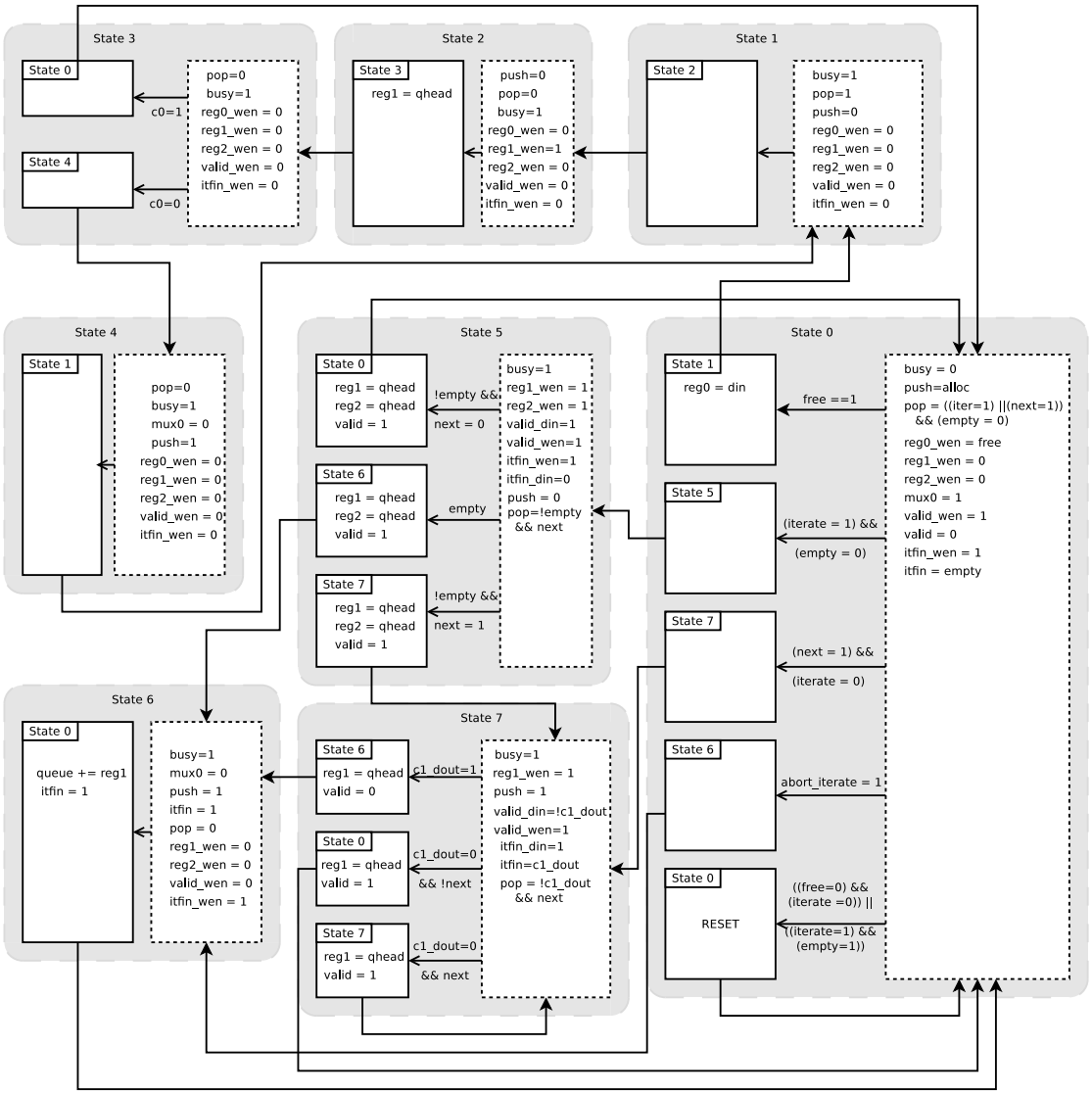


Figure 4.10.: State Machine of the Used List Management Unit

becomes zero, indicating it has completed the operation. After that the slotmanager is ready for the next operation since it is ensured that both lists are synchronized. The iterate, next and abort_iterate signals are directly forwarded to the used list in state zero, the free list is not used for those operations. State three is similar to state five, it also waits until the used list signals that it completed the requested operation. The main difference is that in state three requests for the next used slot in the iteration loop are forwarded to the used list.

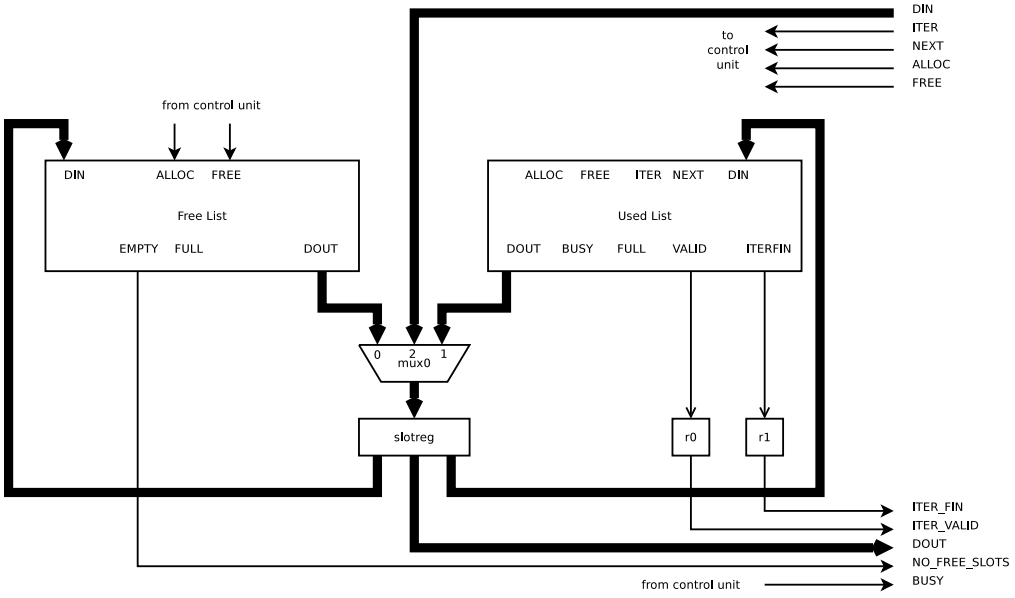


Figure 4.11.: Data Flow of the Slot Management Unit

Note that the values of the iter_valid and iter_fin signals emitted by the used list are buffered in the registers r0 and r1. This is done because the output of the used list during an iteration is delayed for one clock cycle because it is stored in slotreg before it reaches the data out (dout) bus. Since the iter_valid and iter_fin signals indicate if the value at dout is valid or if the iteration is finished it is important that they are also delayed by one clock cycle so that they correspond to the output. Otherwise they would indicate the validity of the output at the next clock cycle.

4.4.2. The Input Consumer

The input consumers task is the aggregation of 16 byte chunks of incoming data. This is necessary because the matching element (the data that is needed to perform matching with) is 64 bit in size. Since we want to have one match at each clock cycle we need this data on a 64 bit bus. Another reason why it makes sense to do this input aggregation in another module is that this allows us to block the state machine of the input consumer while we are waiting for more data. Even though it is mandated to have small queues before the matching unit it is possible that these units become empty while the input consumer is receiving a command that consists of multiple 16 bit blocks, for example an insert command consists of 13 such blocks. However, it would be inefficient if the matcher statemachine would block after the first block

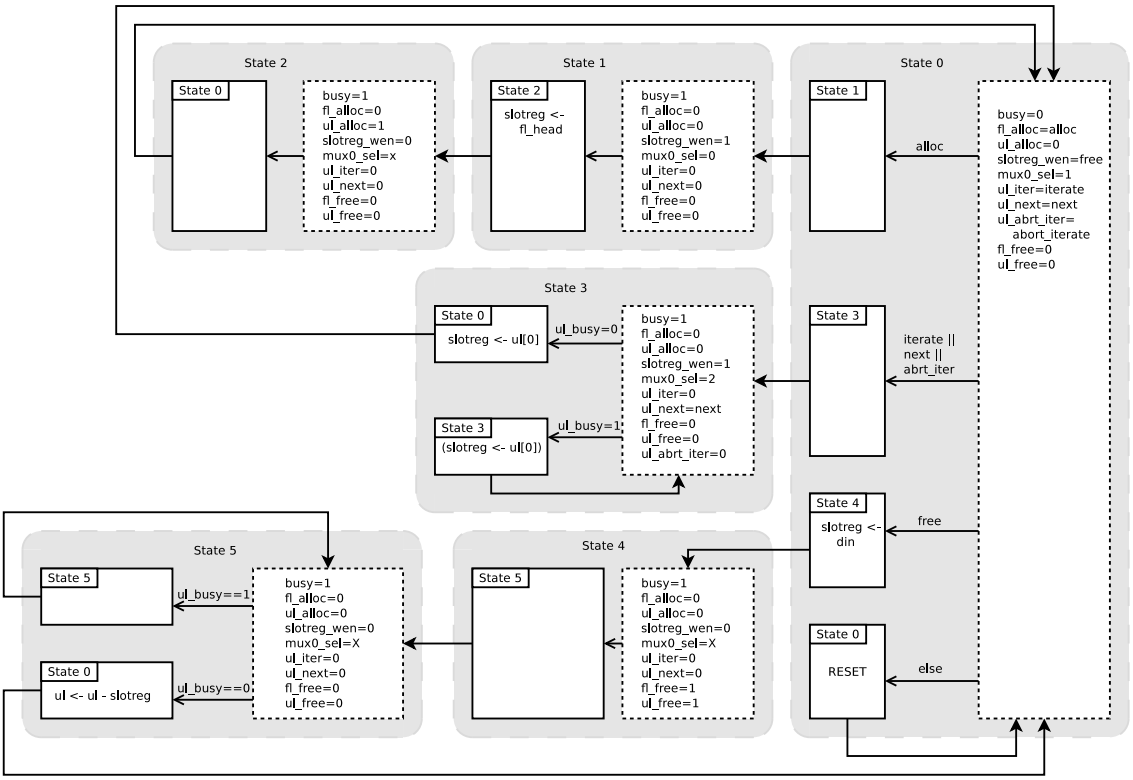


Figure 4.12.: State Machine of the Slot Management Unit

because the second block is not available — during that time other units such as the free list could still perform usefull tasks. The input consumer not only aggregates input it also handles most of the processing of the insert request. When a new slot has to be inserted the input consumer informs the slot manager and waits for the first 64 bit of data, which it writes into the slot memory. It repeats this process until the insertion request is completed. The slot memory is addressed a bit differently from conventional memory: each address consists of two parts, the slot id and the field id. Each slot consists of four fields, the matching element followed by the address and size targeted by the transfer followed by a pointer to the corresponding operation in the schedule.

The input consumers data flow diagram is not shown here because it just consists of four 16 bit registers, each connected to the data in port and the outputs are aggregated into the 64 bit data out bus. Also a four bit register is used to store the opcode of the operation which is currently red. As you can see in the state diagram shown in Figure 4.13 the state machine utilizes two additional registers for counting how many packets of a larger request (insert request) have already been received.

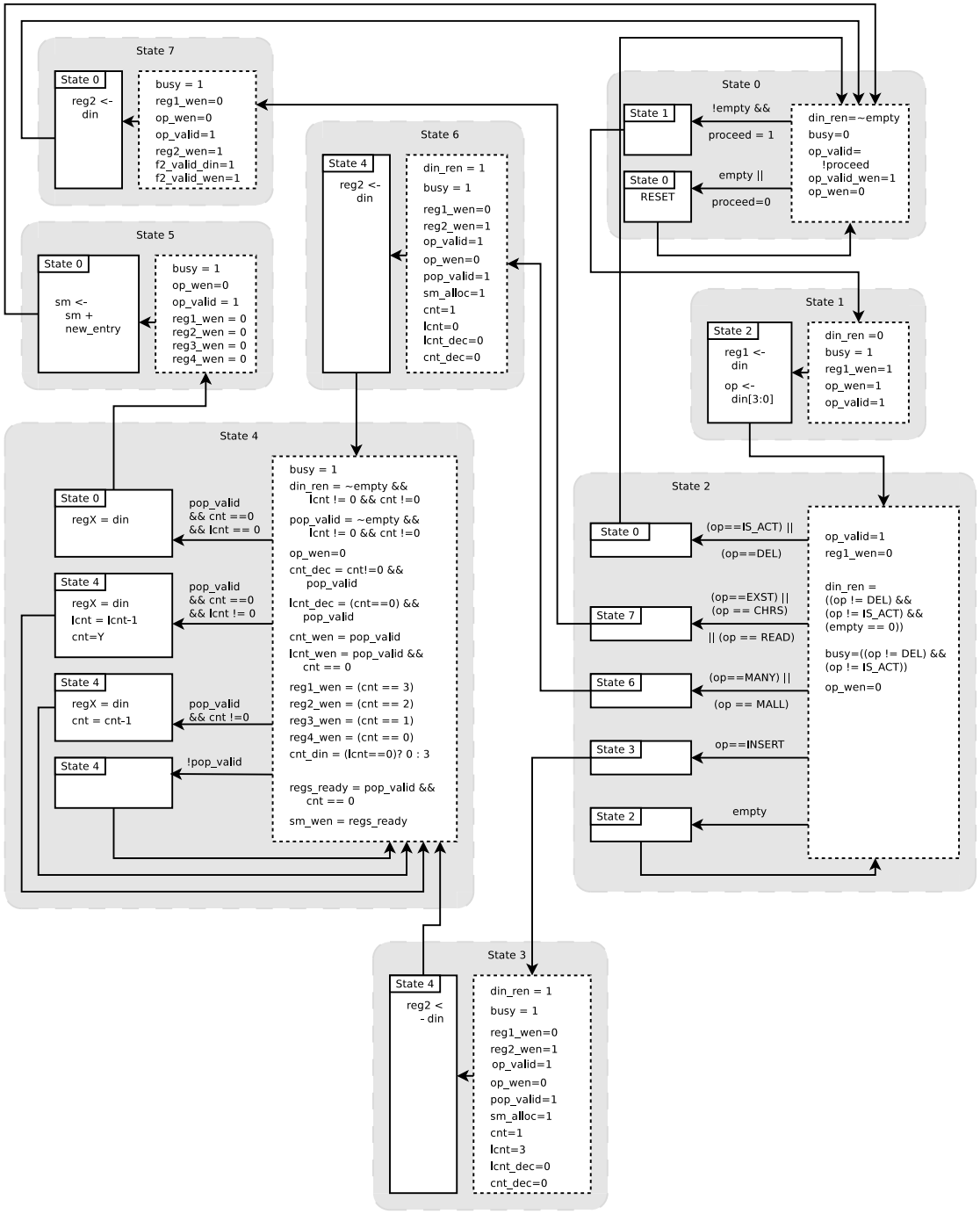


Figure 4.13.: State Machine of the Input Consumer

In state zero the input consumer checks if there is data available from the input queue. If yes the data is popped from the queue. All queues in our design have a one cycle read latency, so in state one the first 16 bit chunk of the new request is written into the first register (reg1) and the first four bit, which define the type of request are stored in the op register. In state two the opcode register contains valid data. This is indicated to the parent unit by setting the op_valid signal to one. In that state the state machine branches, based on type of the request which is read. Some requests such as ISACTIVE or DELETE do not need further processing by the input consumer and consist of only a single 16 bit packet. Therefore, if the contents of the op register is equal to the opcode of ISACTIVE or DELETE, the next state is zero, where the input consumer blocks until the parent unit allows the processing of the next request by setting the proceed signal to one. The proceed signal is required because the parent unit might need the contents of the request for some number of clock cycles after it was read. Without the proceed signal the input consumer would start to read the next request if one is available in the input queue, thereby overriding the current command, making it unavailable for other functional units. Other commands such as EXISTS, CHANGE_REMOTE_SLOT and READ are 32 bits in size, therefore the input consumer has to consume one more 16 bit chunk in this case. This is done in state two if the data is available (the empty input signal is zero), if not we loop in state two until it is. In state seven the second chunk of those requests is written in the appropriate buffer register, and in the next clock cycle the input consumer can potentially read the next command. For the remaining commands the input consumers state machine has a "read loop" built-in in state four: the two counter registers cnt and lcnt are used to specify this read loops behavior: cnt indicates how many more 16 byte chunks should be read to fill a 64 bit field which can be written into the slot memory, while lcnt specifies how many fields are left to be red (only the INSERT command uses this since it is the only command which is bigger than 64 bit). The counter registers are loaded in the states three and six. The actual reading, and in case if the INSERT command also the insertion of the new slot into the slot memory, is performed in state four, in which the state machine loops until the data is read completely.

4.4.3. The Output Generator

The output generator has several tasks to fulfill. It gets informed about the currently processed request by the mode signal, which contains the opcode of the current request and the "phase" of the request processing. The phase can have three different values: WAIT, MATCH, and FIN. During the wait phase the request processing did not start yet because the input consumer is not finished reading the complete command packet. During the match phase the request is currently processed, for

example for the matching commands MATCH_ALL and MATCH_ANY that means that the used list entries are currently being iterated over and the output generator has to record the ids of the slots for which a match occurred. The output generator is informed about matches by the match signal, which is set to one if a match was detected. The queue in the output generator unit has the purpose of temporarily storing the respective slot ids until the response packets for those matches have been sent to the response queue. In the fin phase the iteration is completed and the parent unit is waiting for the output generator until it pushed all response packets into the results queue. Each response packet has a unique header which identifies the response type. These headers are available at mux0. Note that the multiplexer mux0 drives the first four bits of the input of mux1 at selection zero, while the bits five to fifteen are driven by the register reg4 which contains the slot id field of the response packet.

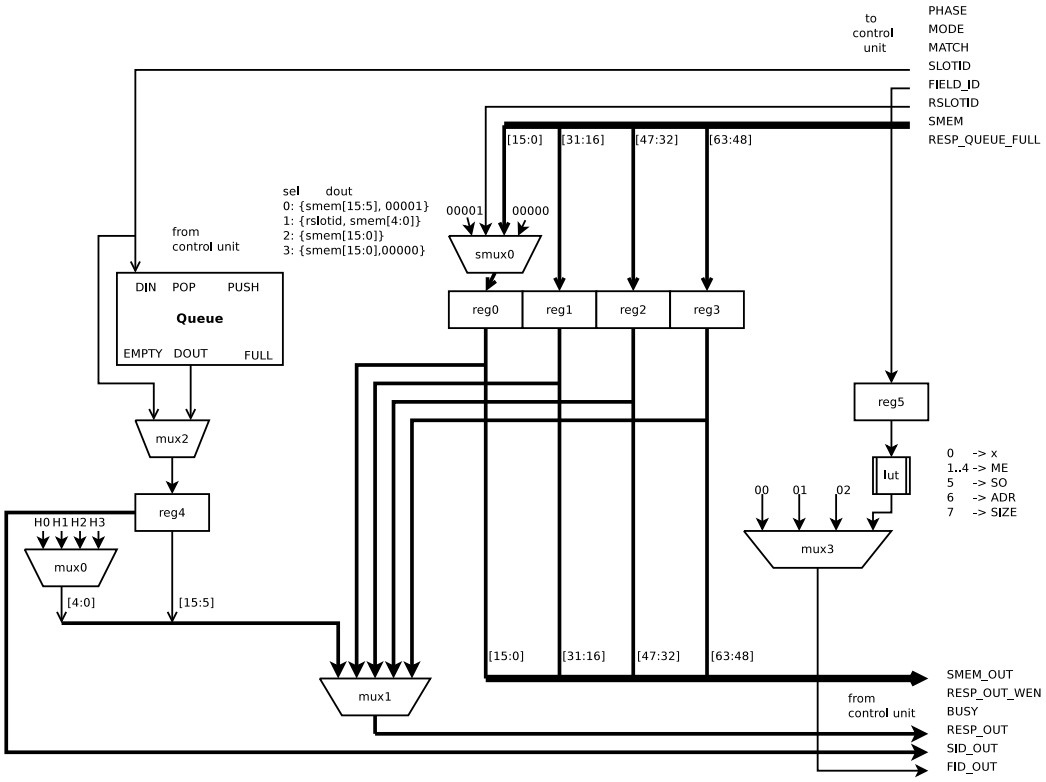


Figure 4.14.: Data Flow of the Output Generator Unit

The output generator also needs to change the ACTIVE bit in matched slots. Since the slot memory has a data width of 64 bit it four 16 bit registers are needed to temporarily store the first field of a slot, change the ACTIVE bit via the smux0 selections zero and three. Note that smux0 is not a multiplexer as it not only selects between inputs, it rather concatenates different inputs based on the selection, as

indicated in the data flow diagram. The output generator also implements the READ command, the requested field is stored in register reg5. The lookup table lut maps the field id supplied by the user to field ids used by the slot memory. For the creation of the match responses the output generator also needs access to most of the fields in the matched slot, therefore mux3 enables to set the fid_out signal to the required values. The multiplexer mux1 drives the response output bus, it enables the splitting of the 64 bit slot field into 16 bit packets.

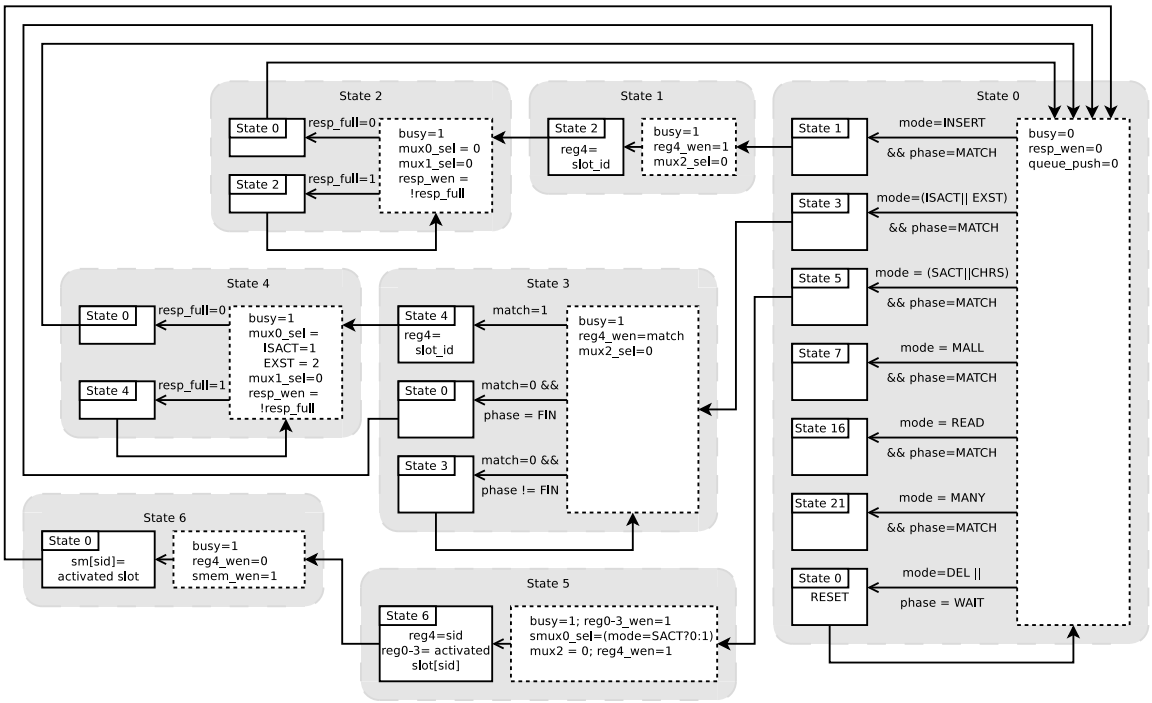


Figure 4.15.: State machine of the Output Generator Unit for the commands INSERT, IS_ACTIVE, EXISTS, SET_ACTIVE, CHANGE_REMOTE_SLOT

The state machine that governs the actions of the output generator is too complex to fit in one picture, therefore we show the statemachine in two parts: Figure 4.15 shows the part of the state machine that implements the commands INSERT, IS_ACTIVE, EXISTS, SET_ACTIVE and CHANGE_REMOTE_SLOT the implementation of the commands MATCH_ANY, MATCH_ALL and READ is shown in Figure 4.16. After a reset the statemachine loops in state zero until the match phase for a command starts. The delete command is ignored by the output generator, since this command does not trigger a response. For the insert command the match phase starts after the slotmanager unit has found a free slot. The id of that slot is assigned to the slotid input bus of the output generator. In state one this slot id is stored in the register reg4. In state two the insert response is formed by concatenating the response header H1 with the slot id in reg4 and assigning it to the resp_out bus. If the response

queue is full the output generator loops in this state until the response could be written. The match phase for the commands is_active and exists is similar: The match testing unit is checking if a slot is marked as active and, if so, indicates this by setting the match input signal to one. If this happens the corresponding slot id is again stored in reg4 in state three so that the response packet can be emitted in state four. If the parent unit indicates that the testing is over by changing the phase input bus to the FIN value the next state is state zero and no output is generated. The output generator loops in state three until one of those two conditions is met. For the commands set_active and change_remote_slot the output generator expects the first field (which contains the matching element) of the slot that should be altered on the smem input bus. The data is written in the four register reg0 to reg3. In state five the first 16 bits of the slot are altered as requested while the slot id is stored in reg4. In state six the altered data is written back into the slot memory. After that the processing of those commands is finished and the output generator resumes into state zero.

The match_all command is the most complex one to implement for the output generator. First all matches have to be gathered in the queue. This is done in state seven. If the parent unit indicates that the matching process is finished (the phase signal becomes FIN) it is checked if the queue is empty in state seven. If yes, no match occurred in the previous matching phase, therefore no output has to be generated and the state machine returns to state zero. Otherwise the slot id stored at the queue head is popped from the queue and written into reg4 in state nine. In the next state, state ten, the first part of the response is emitted. However, the response packet for a match_all request also includes more data then just the slot id. The additional data is written to the resp_dout bus in the loop consisting of the states eleven to thirteen. The lcnt register is used as the loop counter. After the response packet was generated the respective slot also has to be marked as inactive. This is done in the states fourteen and fifteen. After that the state machine goes back to state eight to process the next matching slot, if one is available. The match_any request is less complex, since at most one match has to be expected and therefore the queue is not utilized, the slot id of a match (if one occurred) is directly written to reg4 in state 21. The match_any command marks the matched slot as active. Therefore the first field of the matching slot, which contains the active flag, is read from the slot memory in state 22 and modified (activated) and stored in registers reg0 to reg3. The modified field is written back into the slot memory in state 24. This buffering is not strictly necessary, however, without it we would read data from the slot memory, modify it with combinatorial logic and write it back to the slot memory in a single cycle and thereby create a very long critical path which would limit the maximum clock frequency severely.

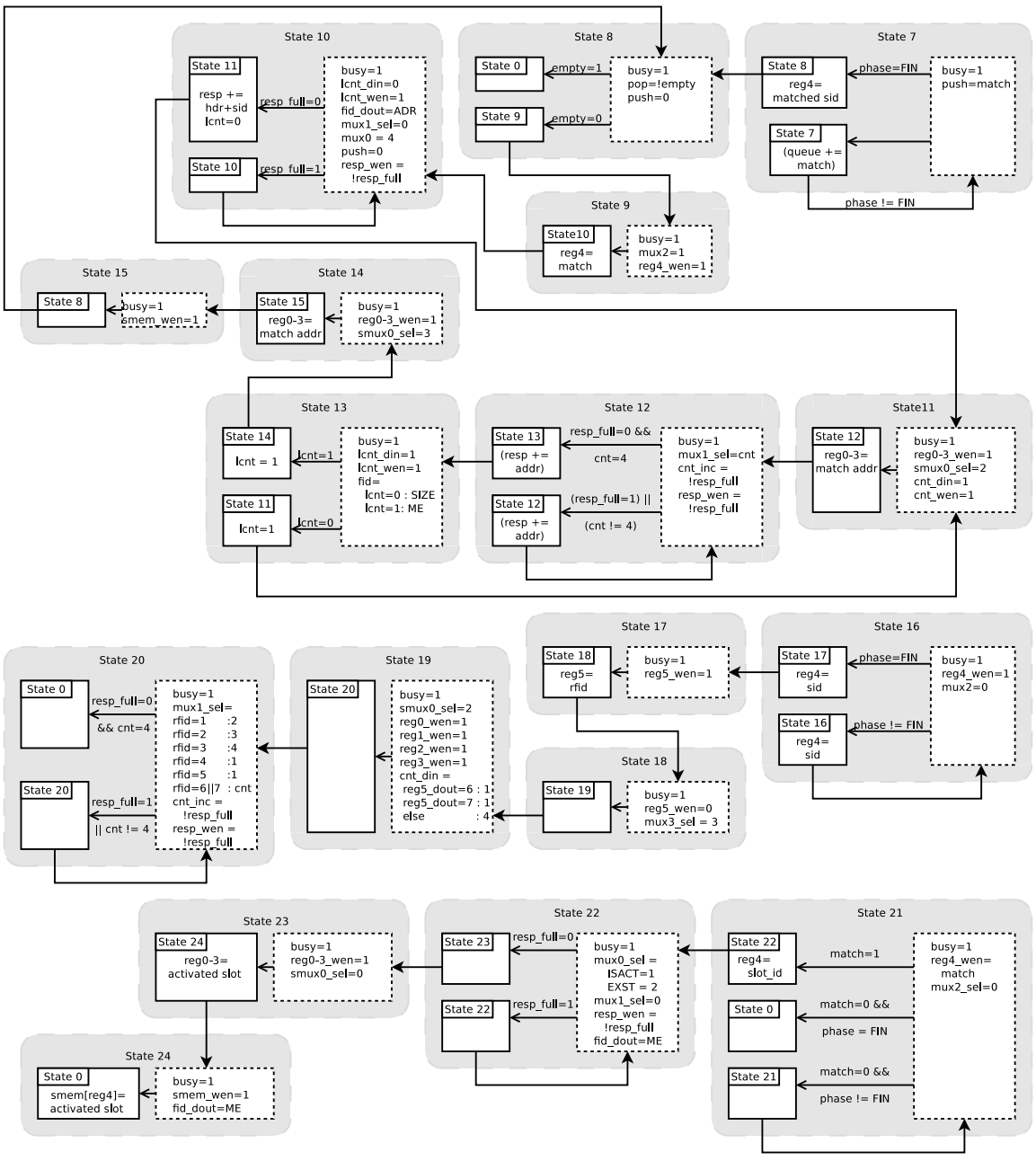


Figure 4.16.: State machine of the Output Generator Unit for the commands MATCH_ANY, MATCH_ALL, READ

The read command is implemented as follows: in state 16 the slot id of the slot from which data was requested is saved in reg4. The read command has no match phase, so the state machine loops in that state until phase becomes FIN. Then the field id is stored in reg5. The field id supplied to the matching unit is decoded by the lookup

table lut and assigned to the fid_out bus, which is connected to the slot memory. In state 19 the data is stored in the registers reg0 to reg3. After the data was stored the state machine loops in state 20 and writes the into the response queue in 16 bit sized chunks. After the last chunk was written the read command is complete and the next state is state zero.

4.4.4. The Matching Unit

We have described all major components of the matching unit above. We will go on to explain how those functional units work together in our implementation. The data path of the matching unit is depicted in Figure 4.17. The input consumer is the only unit that handles input data. It buffers up to 64 bit of data (the size of a matching element) and makes this data available to the compare unit, which uses it to decide if the data read from the slot memory and the data buffered by the input consumer match under the current matching criteria. The input consumer also has to be able to request new slots from the slot manager, and also to request that a slot is deleted. For this purpose bits 5–15 of the input consumers output are also connected to the slotmanagers data in bus. When processing an insert request the input consumer is also responsible for writing the data into the slot memory. Therefore the input consumers output also has to be connected to the slot memorys data in port and the input consumer can drive the slot id, field id, and write enable signal of the slot memory via the multiplexers mux0, mux2 and mux1. Those multiplexers are necessary since the input consumer is not the only functional unit that needs to write into the slot memory. If a match occurred it is the output generators responsibility to activate or deactivate (depending on the matching mode) the matched slot. Since the matching process is fully pipelined (one match comparison is performed per cycle) the registers reg0, reg1 and reg2 are needed to buffer the slotmanagers output so that the slot id signal always corresponds to the match signal at the output generator and the iterate_valid signal always corresponds to the data_in_1 signal at the compare unit.

For brevity we will not show the entire state machine of the matching unit. An example state machine which only implements the INSERT command is given in Figure 4.18. After a reset, the matching unit blocks until the input consumer signals that the first 16 bit chunk of a request has been read an the data at the input consumers opcode output bus (op) is valid by setting op_valid to one. In state one it is important to set the proceed input signal of the input consumer to zero, otherwise it could read the next command before other units have finished processing the data in its temporary registers. As described in Section 4.4.2 the input consumer handles the insert command almost entirely. However, the insert command triggers a response

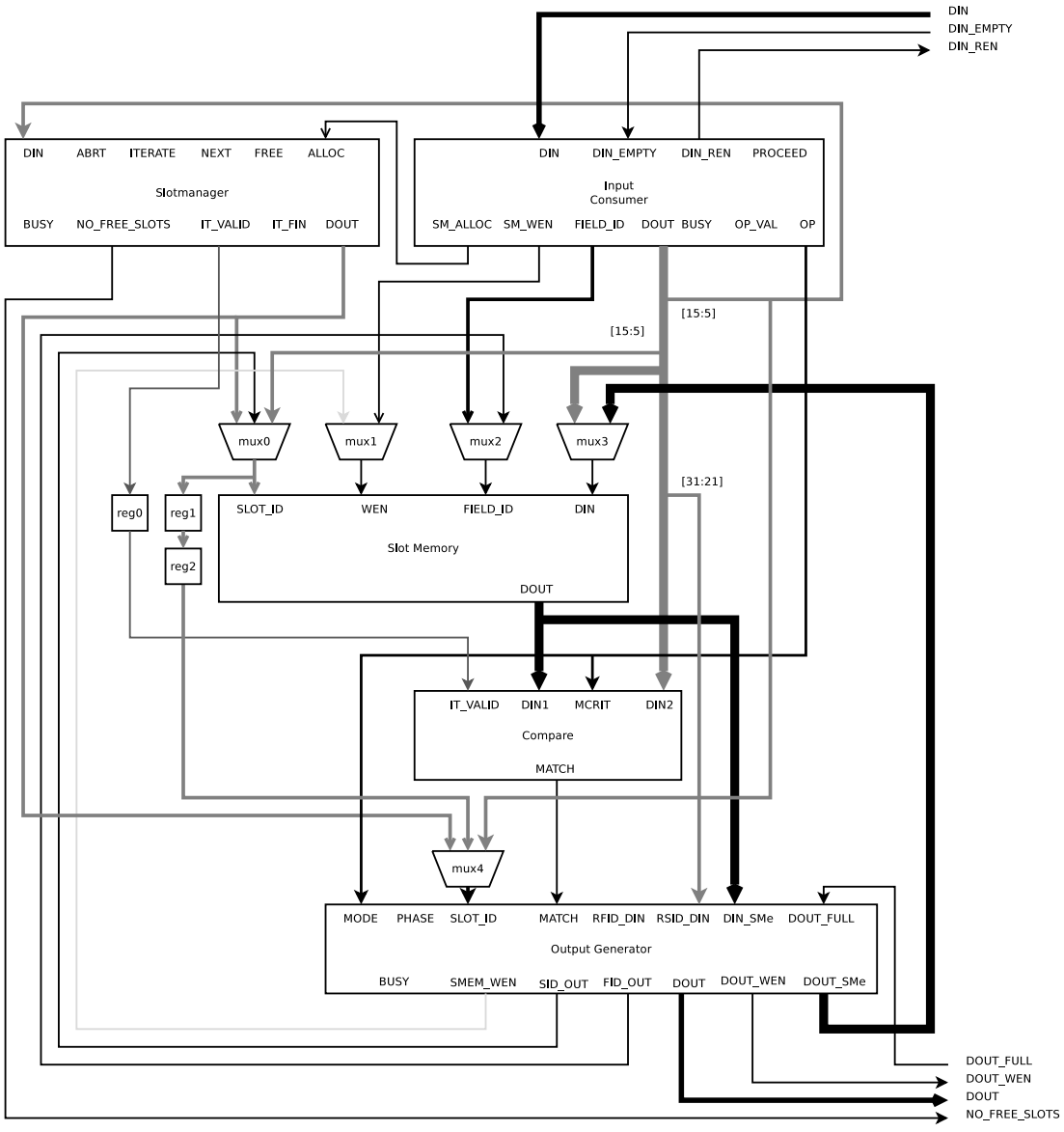


Figure 4.17.: Data Path for the Matching Unit

from the matching unit, the output generator has to emit the slot id of the newly created slot. The slot id is available as soon as the slot manager unit indicates it is ready. Therefore the state machine of the matching unit blocks in state one until the slot manager indicates it is finished processing the last request. While the select signal of the multiplexer mux4 is set to zero, the output generator can create the response packet. We signal the validity of the input data to the output generator by

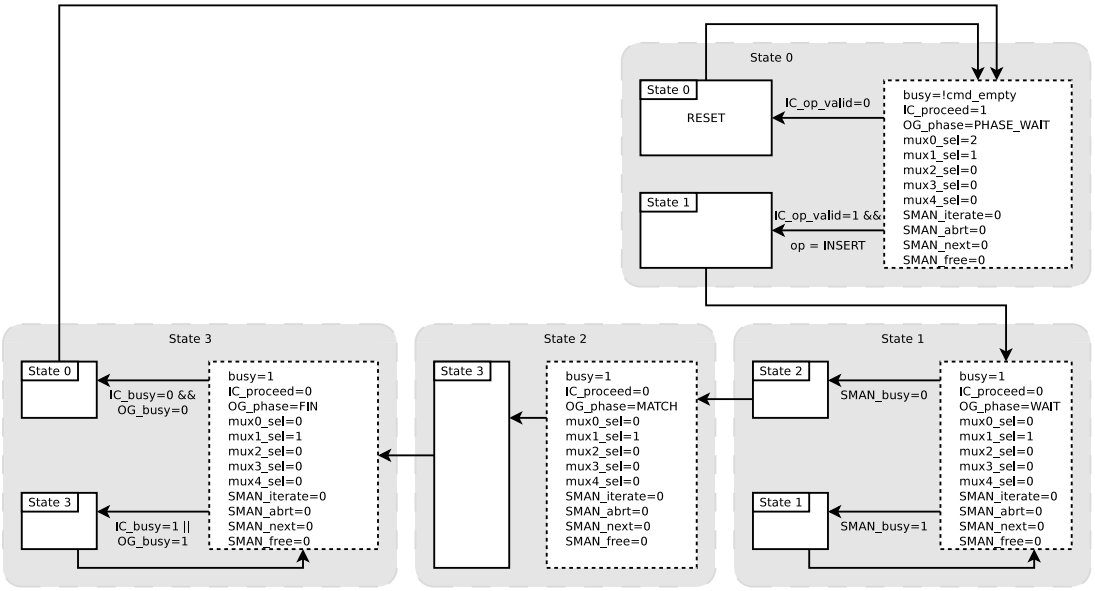


Figure 4.18.: Part of the State Machine for the Matching Unit (INSERT Command)

setting phase to MATCH for one cycle in state three. In state four the matching unit loops until both, the input consumer and the output generator, indicate that they finished the current task by setting their busy output signals to zero. If both units are finished the matching unit is ready for the next command and returns into state zero.

4.5. Slot Management Unit for Non-synchronous Transfers

The non-synchronous protocol is much simpler as the synchronous protocol, as no matching is required. Nevertheless, if a new RDMA operation is started the GOAL unit has to store some state information for the transfer. If a RDMA operation finishes this information is used to identify the operation that is responsible for the transfer in the GOAL schedule so that this operation can be marked as completed. Unlike the matching unit, the non-synchronous slot managements unit is not packet based, as it offers less functionality. It is able to store 256 slots of the size 16 bit. If a new slot is added the slot management unit returns the address of a free slot. If no free slots are available this is indicated by the full signal having the value one. The parent unit can also delete a slot by assigning its slot id to the data in bus and setting the delete input signal to one. It can also read the contents of a slot via the

read signal. Once the data on the data out bus is valid this is indicated by the valid signal.

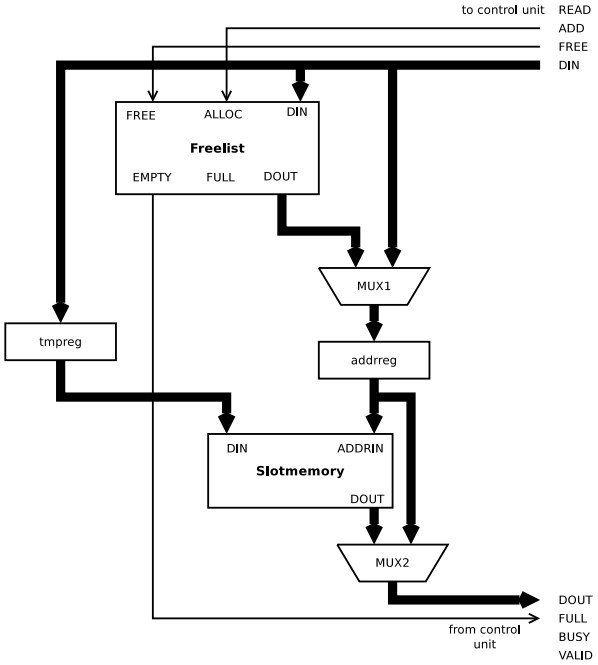


Figure 4.19.: Data Path of the Non-Synchronous Transfer Slot Management Unit

The data path of the non-synchronous slot management unit is depicted in Figure 4.19. We utilize the free list also used by the slotmanager for the matching unit to manage the available slots. Since the non-synchronous slot management unit does not have the functionality to iterate over all used slots a used list (as used in the slotmanager for the matching unit) is not necessary. Two registers are used in the design of the non-synchronous slot manager: The register tmpreg will temporarily store the content of a newly added slot until it has been written into the slot memory. The register addrreg is used to store the address returned from the free list upon slot insertion. This address is then used by the slot memory to access the slot but also provided on the data out bus so that it can be read by the parent unit.

The state machine of the non-synchronous slot manager is shown in Figure 4.20. Note that the state machine does not have a branch for the free command - the free input signal is connected directly to the free input signal of the free list, so freeing/deleting a slot is handled entirely by the free list. In state zero the state machine will write the data assigned to the data in bus (din) into the register tmpreg if the parent unit requested to add a slot. Add the same time the free list will emit the id of a free slot. It will be written into addreg in state three. In state four the slot content, which

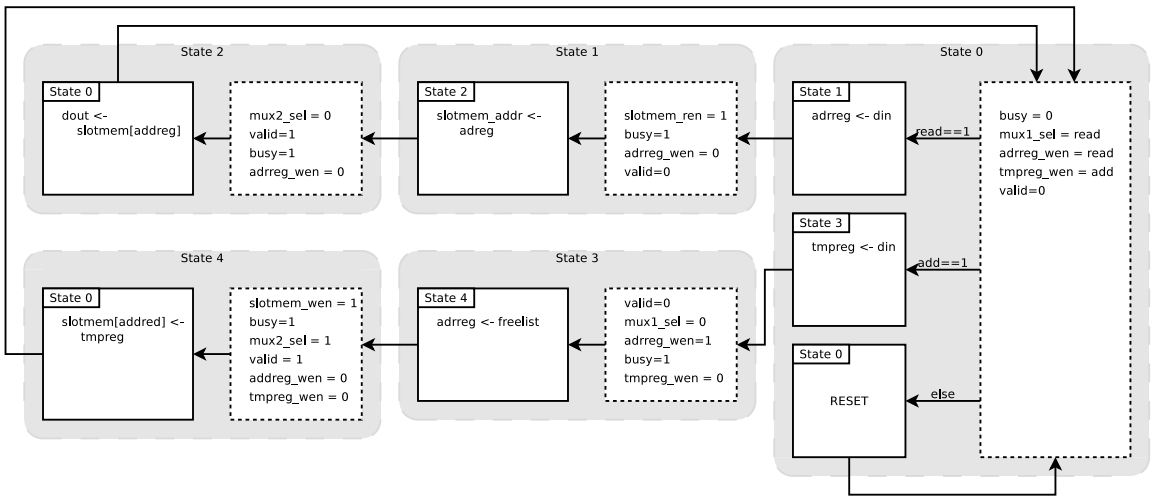


Figure 4.20.: State machine of the Non-synchronous Transfer Slot Management Unit

was stored in tmpreg before is written into the slot memory at the address stored in addreg. The valid output signal is set to one so that the parent unit knows that the address is valid. To implement the read operation the address (id) of the slot that should be read is stored in the addrreg register in state zero. In state one the data is read from the slot memory, which has a read latency of one cycle. In state two the data is available on the dout bus and the valid signal is set to one.

5. The GOAL Interpreter

In this chapter we will describe the Verilog implementation of the schedule interpreter unit. We will give an overview over the design which we built from several smaller functional units. This is neccessary to handle the complexity of such a design but also provides flexibility for future research as modules can be exchanged with optimized implementations as long as they have the same interface.

5.1. Schedule Interpreter Design

The GOAL Interpreter is the main component that will be developed in this thesis. It contains all other units and forms a more complex system using them. It is responsible for the interpretation of the GOAL graph as introduced in Section 2.4, starting of operations and tracking their state until the schedule finishes. Figure 5.1 shows an overview of the main components including the external COMM unit. It is not part of this work, but will be briefly explained in Section 5.2 to provide a better understanding how these components are supposed to interact.

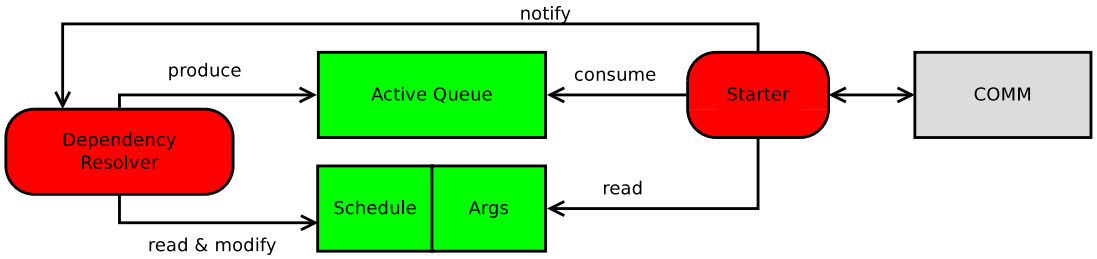


Figure 5.1.: Overview of the schedule interpreter design. Green boxes represent passive storage units, red boxes represent active functional units, arrows represent access. The interface towards the transceivers is not addressed in this work

The units schedule and args are two memory blocks which contain all data directly related to the schedule and its operations. The units which operate on them require

that the memory blocks are correctly initialized before they are started. Correctly means here, that the schedule stored there fits in the schedule memory and that it can terminate as explained in Section 3.4. Also all arguments of operations referenced in the schedule have to fit inside the operation argument memory.

The dependency resolver is the only component that is able to interpret the binary schedule that was designed in Section 3.3. It is started by a notification from another component and searches for operations that have no further dependencies and can be started. The addresses to the operation arguments are placed inside the Active Queue and will be processed by the unit called Starter. The Active Queue is implemented so that it follows the requirements from Section 3.5 to prevent deadlocks and buffer overflows.

The Starter itself is a group of units that can execute different types of operations. In the current implementation, it contains all components to manage transfers based on the protocol developed in Section 3.6 and 3.7 and one component for non-synchroneous transfers. The matching unit developed in Chapter 4 is used as the unit to identify transfers and to store the current protocol state.

5.1.1. The Active Queue

The Active Queue is one of many different queues that are used inside the GOAL interpreter. They connect different units to allow them to receive new data while they are busy or as temporary memory inside a unit to save results that are processed further at a later point. All those implementations share a common design that is only slightly modified to provide a larger queue or a wider element size.

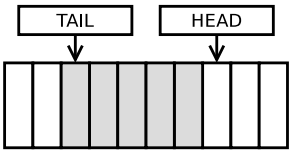


Figure 5.2.: Principle of operation of the active queue. Grey elements are deleted.

Figure 5.2 shows the basic idea that was explained in Section 3.5. The queue is build using one or multiple BRAM blocks that were concatenated as explained in Section 3.3. Simple dual port memory is used to allow concurrent read and write to the same queue. Two registers with the same width as the address port of the memory contain the start and the end of all elements stored inside the queue. The tail register points to the next position where an element is written. A push operation

would therefore write the element from the din port to this location and increases the tail pointer. A pop operation uses the head register to read an element. It will be sent through the dout port to the next unit and the head pointer is increased. When both registers contain the same value, it is not clearly defined whether the queue is full or empty. To resolve this problem, a third register is used to track the amount of elements in the queue and combinatorial logic that generates the full and empty signals from this count register.

The queue itself has a limited functionality and its performance highly depends on the used memory. It provides no internal check for the queue state to prevent that external units write on a full queue or read from an empty one. The units attached to the queue have to check the state before they can access it. It also shows the same characteristics as the BRAM blocks on the Xilinx FPGA. Therefore, a push takes one clock cycle, but a pop two cycle to finish.

5.1.2. The Dependency Resolver

The task of the Dependency Resolver is to manage the state of the schedule and to inform other components about operations which can be started by them. It receives notifications about operations which finished and reads the schedule memory to find them and all their adjacent operations. The dependency counter of these adjacent operations can be reduced because one of their dependencies just finished. A dependency counter of zero indicates that this operation does not wait on any other operation to finish and can therefore be started. The unit does this by reading the address of the argument and appending it to the Active Queue.

Figure 5.3 shows the components of the dependency resolver and the connections to the schedule memory, the Active Queue and the input that receives the address to the finished operation. The read and write ports of the simple dual port memory are directly connected to the Dependency Resolver. This is done to allow simultaneous read and write operations by the unit. Signals and the control unit are removed from the diagram to reduce its complexity.

The register reg0 is used to store the address of the finished operation and in later cycles the position of the adjacent operation counter. The value in register reg1 points to the memory range that contains the addresses to the adjacent operations and register reg2 the amout of remaining adjacent operations. Both registers are used together to loop over the adjacent operations and change their dependency counter using reg5 as temporary register. The position of these operation will be loaded into register reg3 and reg4 receives a pointer to the address of the operation arguments.

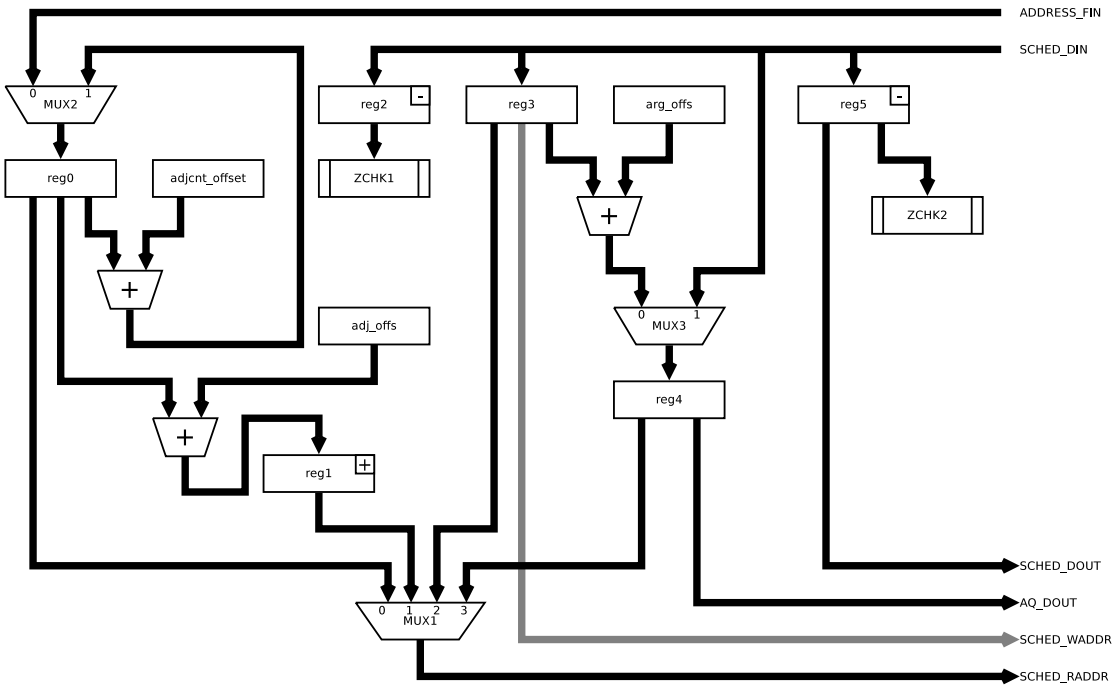


Figure 5.3.: Data Flow of the Dependency Resolver and connections to external components

A register with a small box in the upper right corner defines the special functionality of it. A plus sign marks a register capable of incrementing its content by one and a minus a register that can decrement the stored value. A box with a double border on the left and right side is combinatorial logic that can for example compare the input with zero or add two values together.

The state machine of the Dependency Resolver shown in Figure 5.4 is slightly different compared to the state machines in Chapter 4. Dashed rectangles only contain information about registers, combinatorial logic, data buses and connections between them. Multiplexers are not shown because connections already define how the select signal of the multiplexer must be set to connect these components. For example the state 3 attaches reg1 with the schedule read port address input. Therefore, multiplexer mux1 select has to be set to one during this state. Red names are variable names used to create named connections in context of a single state. The name adj-cnt is such a connection for state 3 and defines that the output of the schedule data bus will be used in this state and is connected to reg2. The solid rectangles are actions which happen during a posedge. For example, State 7 has multiple actions such as the storing of the schedule output in reg4 or the decrementing of the value in reg4. States with the word “RESET” are the states the unit will enter after a reset signal.

This state is, when not otherwise defined, the only state that has the busy signal set to zero.

The type of state diagram used in this chapter is less precise than the previous one. This makes understanding the different actions done in a state easier, but also requires more attention during the implementation in Verilog. An example is register reg0 that is only used in state zero, one and two. The write signal for this register can be "dont-care" for state two and higher. This is not obvious by looking at a single state, but when checking all other states.

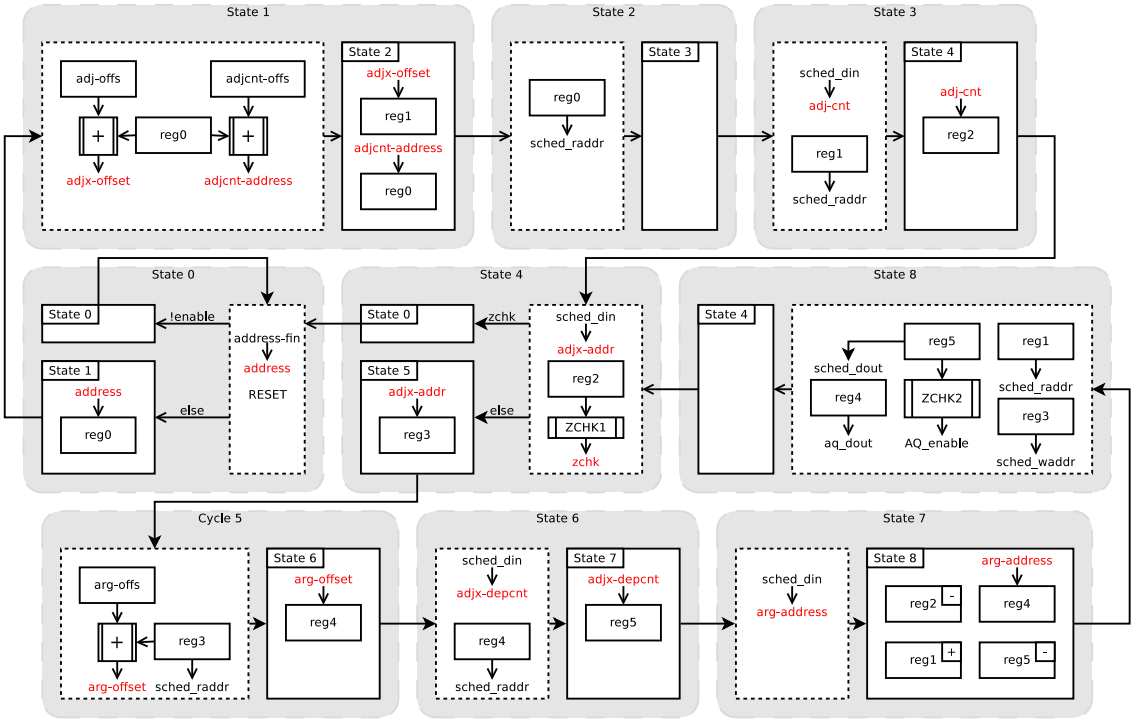


Figure 5.4.: Finite State Machine for the Dependency Resolver

The Dependency Resolver waits in state zero for the address of a new finished operation and stores it reg1. It is used in state one two calculate the pointer to the address of the first adjacent operation address and pointer to the number of adjacent operation of this operation. The unit stores this in the register reg0 and reg1 to reduce the net and routing delay. State two can now start a read of the count from the schedule memory and saves the result in state three to register reg2. At the same time it also starts the read of the first adjacent operation address. It can happen that an operation has no adjacent operations and the returned value cannot be used This will be detected in state four when the counter in reg2 is compared with zero. The Dependency Resolver will directly go to state zero in this situation and drop all read information. Otherwise it will save the address in reg3 and start the loop body with

state five. This address points directly to the dependency counter of the adjacent operation that must be decremented. A read for this counter is started and the same address is incremented by one to get the address of the operation arguments. State six saves the counter to reg5 and immediately starts the read using the pointer in reg4. The number of still remaining adjacent operations and the dependency counter is decremented in state 7 while the pointer to the next adjacent is is prepared and the operation argument address is saved in reg4. This new values can be used to finish this adjacent operation in a single cycle and also to start the read of the next adjacent operation address in state eight. The new dependency counter is written to the schedule memory while its value is compared with zero. The adjacent operation argument address will only be written to the Active Queue when this comparison returns true. State four can now again start the loop body or return to state zero when no more adjacent operations are available.

When finishing an operation which has n adjacent operations, the Dependency Resolver needs $5(n+1)$ cycles. In case that all adjacent operations will be started, the first operation argument address will be written after 10 cycles to the Active Queue and 5 more for each additional one.

5.2. Transceiver Interface

Communication of the GOAL schedule interpreter with external resources is realized through a communication module developed in another thesis. A preliminary specification was used to develop the protocol units. It provides support for small transfers not greater than 64 Bytes which can be sent directly to another local or remote unit. Transfers larger than 64 Bytes are called bulk transfers in context of the COMM interface and need to store transaction information in all communication modules involved. The unit will handle this management work, but may block when not enough buffer space is available to store the information. This information can also include extra user defined bits which get returned in the notification message after the transfer finished.

The unit has separate interfaces for control commands and data exchange. The data interface shown in Figure 5.5 must be used to send or receive data directly from the GOAL interpreter. Transfers cannot be started directly over the interface and the GOAL unit has to answer to the request through the read and write ports. Both ports have a 64-bit wide address and data bus similar to simple dual port ram. A write is only triggered when write_valid is one and reads only when read_request is

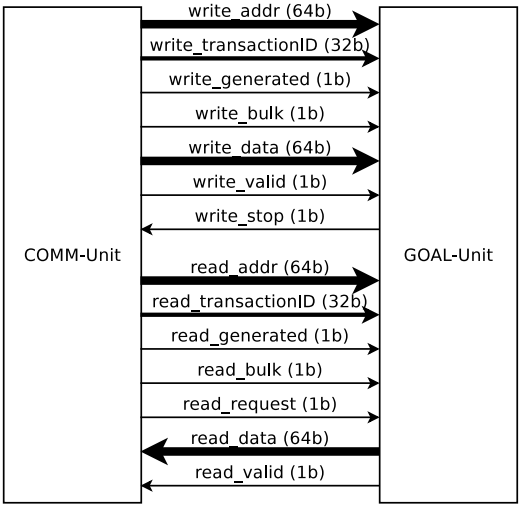


Figure 5.5.: Connections between the COMM Data interface and the GOAL unit

one. The GOAL unit does not have to be able to process all request directly but can postpone the request by using either write_stop or read_valid.

The communication unit also provides additional signals like transactionID, generated and bulk for both read and write ports. These information can be used to find corresponding transfers and act according to private data not stored in the communication unit. They are only optional and not used by the current GOAL interpreter design.

The transfers are started and controlled over a command interface shown in Figure 5.3. The GOAL unit can send commands using a 64-bit bus and the COMM unit can also return replies through the 64-bit notification bus. Both units can block any new commands or replies by using the wait_b for information related to bulk transfers and wait_nb for information related to non-bulk transfers. Otherwise the remote unit has to read the new data after the corresponding valid signal is one. This signal has to kept high during the complete data exchange and therefore the exchange cannot be paused.

The only command used by the GOAL unit is DATA_MOV which is responsible for transferring data between different end points. This can be a local or remote units and units can either be the GOAL unit or host memory. Non-bulk transfers are used to transfer smaller control messages from a local to a remote GOAL unit. The only information sent by the GOAL unit over the command interface are a command packet header, size of the transfer and the peer id of the remote unit. Bulk transfers are used to get data from remote memory to the local memory. They also need local and remote address information which could be omitted for the non-bulk transfers.

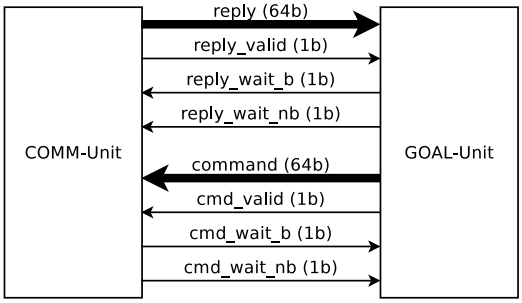


Figure 5.6.: Connections between the COMM Command interface and the GOAL unit

5.3. The Starter

The Starter shown in Figure 5.7 has the task to retrieve operations that can be started and process them depending on the operation type. This includes synchronous send and receive operations, but also RDMA and operations without data transfer to external components. Each type is handled by a specialized sub-unit of the so called Protocol unit.

All now independent actions will be passed to the starter from the dependency resolver through the active queue. Only pointers into the operation argument memory are stored in the active queue and another unit has to decide for which protocol sub-unit the next operation is suitable. A currently busy sub-unit which has to process the next operation would also block other units when the assignment to the sub-units would not be buffered and no random access is allowed in the active queue. Therefore, a unit called Sorter is introduced which only consumes an address from the active queue, reads the corresponding type from the operation argument memory and saves the address in the corresponding protocol sub-unit queue. A sub-unit queue has the same design as the active queue from Section 5.1.1, but the size of the queue can be freely chosen and not all operations in a schedule have to be stored inside the protocol sub-unit queue. The reasonable size is limited by the maximum number of elements in the schedule which is also the number of elements in the active queue. This results from the fact that the sorter only takes operations from the active queue and does not generate new operations.

The protocol units can now consume entries from their protocol sub-unit queues whenever resources are available to start new operations and the queues are not empty. The limiting resources are currently the matching units for the synchronous send and synchronous receive operation protocol unit explained in 4. They only have a

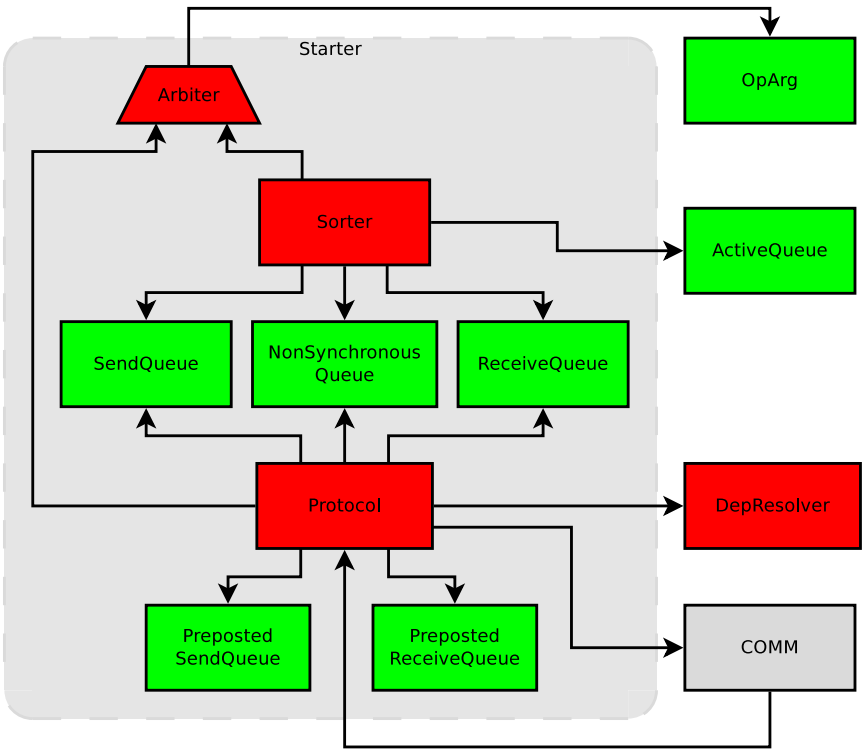


Figure 5.7.: Overview of the Starter design. Green boxes represent passive storage units, red boxes represent active functional units, arrows represent access.

limited amount of slots and therefore only a limited number of operations can run simultaneously.

All protocol units need access to the operation argument memory to read the parameters of send and receive operations. They also need to send notifications about finished operations to the dependency resolver. Arbitration units have to be be used to prevent concurrent access to these units. The protocol units and the sorter may only assume that the submitted data are valid when the arbiter informs them about it. Therefore, the single cycle read and write operations as used in the dependency counter are not possible with the arbiter when another unit also tries to access the same resource.

The protocol unit also has the task to manage the concurrent read and write accesses to and from the COMM unit. This includes fast response time on incoming packets to prevent blocking of the COMM unit and uninterrupted transmission of command packets for the synchronous protocol.

5.3.1. Starting Operations

Each protocol sub-unit as shown in Figure 5.8 can decide in wait cycles whether it can start new operations depending on the current slot usage and elements in the sub-unit protocol queue. The non-synchronous protocol has to check its internal operation memory and receive/send protocol has to check the corresponding signal of the external matching units Preposted Send-/RecvQueue.

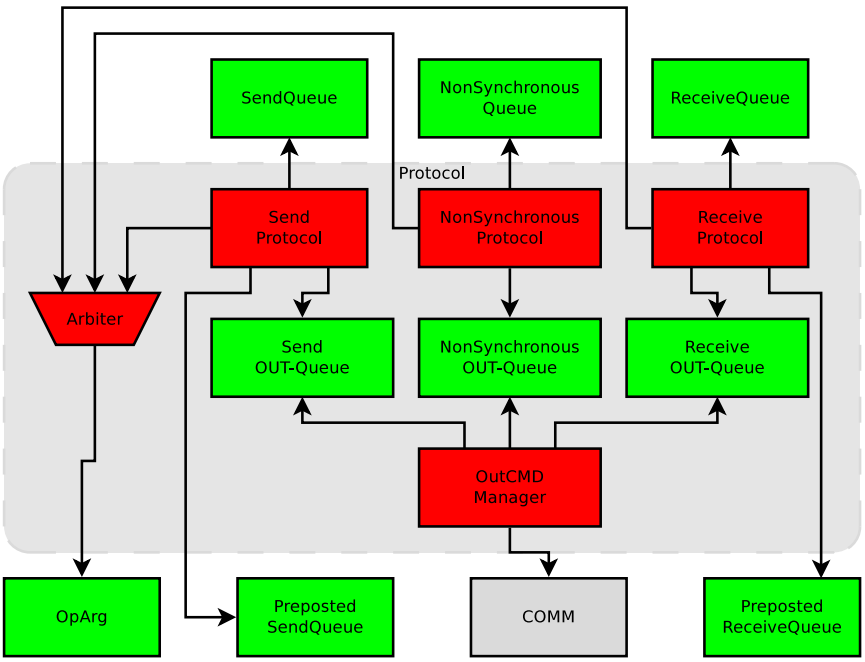


Figure 5.8.: Interaction between the protocol unit and external components during sends

The state management and message generation is independent from other protocol units and no unit will try to access another one. Therefore, the send and receive protocol units can start their transfer by receiving the operation arguments from the operation argument memory and inserting this information in the corresponding matching unit. The matching unit will return the local slotid which has to be used as part of the specialized ready messages.

A SND_RDY as shown in Figure 5.9 has to include the local slot id to have a direct access for answer packets, the own hardware address and the actual matching element which contains the communicator id, schedule id and tag. The receiver can directly use this information to find a match. The address and size of the buffer can be used to directly start the matching receive using the RDMA functionality of the COMM unit.

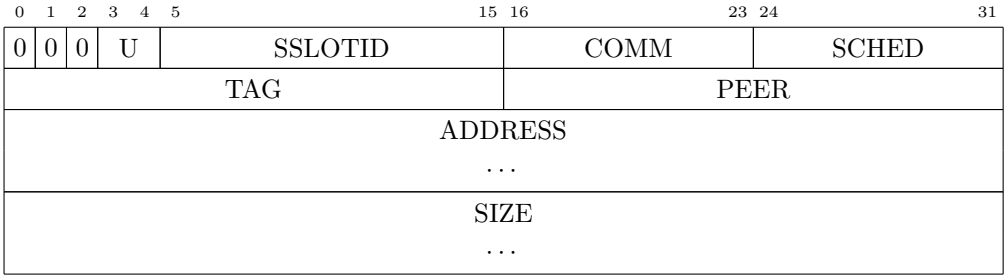


Figure 5.9.: SND_RDY packet format

The receive protocol unit has to create a similar command packet, but can omit the address and size information because the sender is not actively involved in the RDMA transfer and cannot use it to provide further functionality. The packet as shown in Figure 5.10 provides the matching element and the local slot id for the response messages.

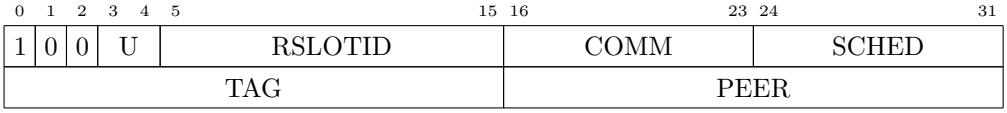


Figure 5.10.: RCV_RDY packet format

Both can start their transmission by writing in the protocol sub-unit specific out-queues. All queues are designed to be able to hold a message for each slot. This prevents a deadlock between the two units which try to send a command packet for a slot when the output queue is full. During this transfer both would not be able to receive new data and thus both units would block and wait for their remote unit to accept messages again. Using a buffer large enough to hold messages for all slots allows the protocol sub-unit to create new ready packets or send answers to ready packets for each transfer without depending on the readiness of the remote unit. It is still not possible to create all ready messages and react on the ready messages from the remote unit for these slots at the same time. A similar message buffer on the remote host works around that problem. The protocol unit would have for M slots enough room for at least $2M$ messages and could save everything in it when the remote protocol sub-unit is not able to process the incoming packets.

The non-synchronous protocol sub-unit can just save the DMA_GET packet as shown in Figure 5.11 in the Non-synchronous OUT-Queue. Extra buffers to prevent deadlocks between the remote units are not necessary because the RDMA GET has no protocol sub-unit on the remote which would also want to communicate with this unit. It can still be useful to add a minimal buffer to overlap the time the unit has to

wait for the access to the operation arguments memory with the time it takes until the data for the transfer are accepted by the shared COMM unit.

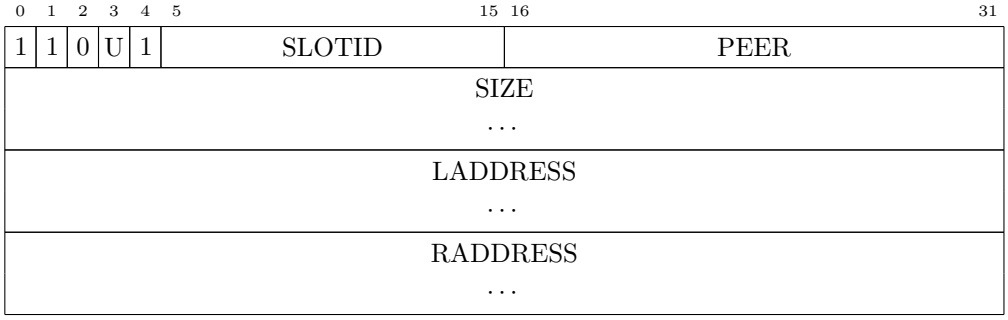


Figure 5.11.: DIRECT_DMA_GET packet format

The Outcommand Manager has the task to read the messages from all protocol sub-unit queues and create DATA_MOV request for the COMM unit. All command packets are transferred as non-bulk messages between two user modules and the actual RDMA GET initiates a bulk transfer between memory. The COMM unit informs the protocol units about finished transfers through a notification packet.

5.3.2. Processing Incoming Packets

Incoming messages from the COMM unit have to be accepted as fast as possible to free resources in the COMM unit again. The IN-Filter tries to fetch the data and push it in so called inqueues as shown in Figure 5.12. The correct destination and size of a message can be found by looking at the first bits of each incoming message.

Each protocol sub-unit can now operate independently on their inqueue. It is therefore possible to implement the protocol as complete separated units without knowledge of the other protocol sub-units. This decreases the complexity of the state machine and makes special synchronization units unnecessary.

Incoming Send Packets

The send protocol sub-unit is only informing the receive protocol sub-unit about new transfers and answers question about the status of transfers. Only the packet type RCV_RDY shown in Figure 5.10 and the FIN packet shown in Figure 5.13 have to be answered.

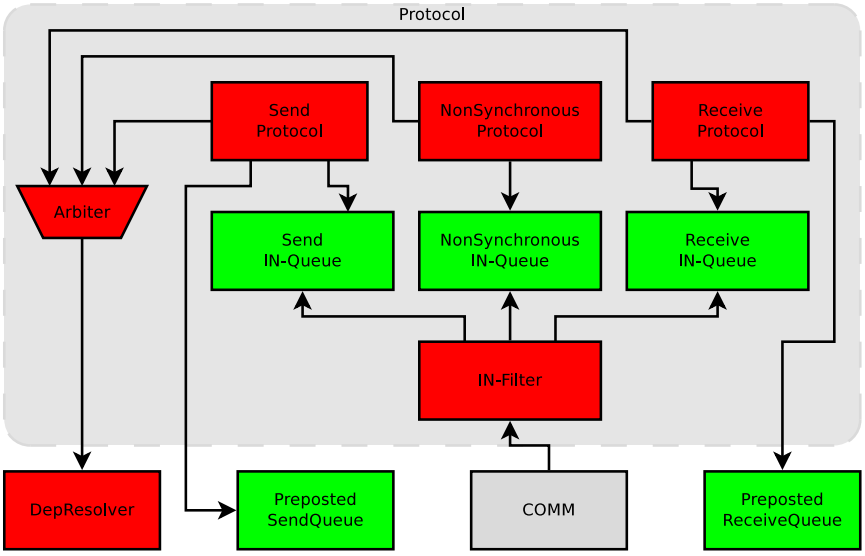


Figure 5.12.: Interaction between the protocol unit and external components during receives

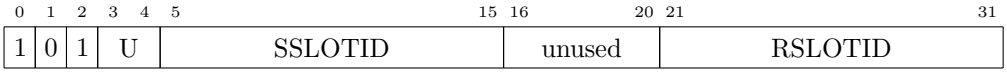


Figure 5.13.: FIN packet format

RCV_RDY The arrival of a RCV_RDY packet informs the sender that the receiver started a new transfer, but does not know the source address and size. The only information useful to identify the corresponding send is the matching element consisting of COMM, SCHED, TAG and PEER. The sender has to retrieve it and ask the preposted receiving queue whether it has a slot with the same matching element which was not already matched. If there is no matching slot then the sending protocol has to return to the initial state and retrieve new messages or start a new transfers.

The returned slot id can be used by the protocol sub-unit to create an answer ACK packet as shown in Figure 5.14 to the remote receive protocol sub-unit. Sender and receiver slot id are used to identify the answer and are copied from the RCV_RDY packet and the returned slot id. Address and size are used by the receiver to send a DATA_MOV command to the COMM unit.

FIN A FIN packet informs the sender that a RDMA transfer has been finished and all information regarding that slot id are not needed anymore. The sender protocol unit can be sure that the mentioned sender slot id is the one he can delete, but still has to check for inconsistency in the preposted sender queue. Sender-side only matched transfers and Sender-side missed matches mentioned in Section 3.7.3 can be detected by getting all unmatched slot id with the same matching element and the slot ids of all matched transfers with the remote slot id retrieved from the FIN packet.

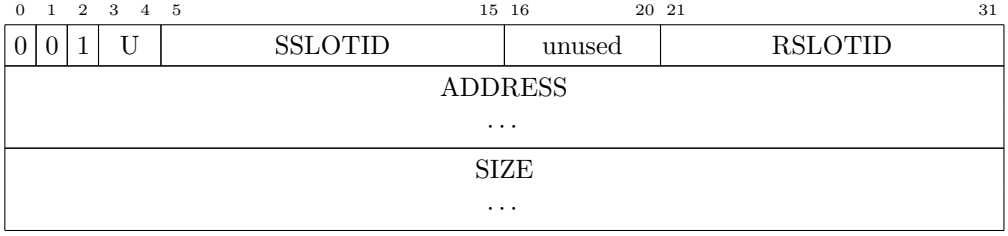


Figure 5.14.: ACK packet format

The send protocol sub-unit has to retrieve important information from the finished slot, delete it and ask the matching unit using the stored information to provide a list with slot ids to be restarted. The unit has to send a FIN_ACK packet as shown in Figure 5.15. The list of slots have to be used to create new SND_RDY packets like the unit did when the operation was started the first time, but this time by starting multiple operations and reading the content of the packet from the preposted send queue. The active flag has also to be resetted on all restarted slot to get again to a matchable state.

The schedule offset which was stored in the preposted send queue slot has to be sent to the dependency resolver. All adjacent operations with a dependency counter of zero will automatically be started.

Incoming Receiver Packets

The receive protocol sub-unit gets replies from the remote sender protocol sub-unit and notifications from the COMM unit. SND_RDY as shown in Figure 5.9, FIN_ACK as shown in Figure 5.15 and DMA_FIN as shown in Figure 5.16 are the only packets the unit has to process.

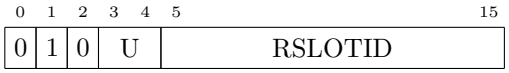


Figure 5.15.: FIN_ACK packet format

SND_RDY A SND_RDY packet informs the receiver that the sender started a new transfer and the receiver can use an RDMA transfer to get the data from the peer and address stored in the packet. The receive protocol unit has to find a matching slot id in the preposted receive slot which was not already started and check if no other started slot already has the same peer and sender slot id. A failure will drop that packet and allows the unit to start new transfers or process new packets.

The returned slot id is used to store the sender slot id and retrieve the local address and size of the transfer. A DMA_GET has to be created to initiate the RDMA Get. The structure of a DIRECT_DMA_GET as shown in Figure 5.11 is used with the identifying bits `011X0`. The address fields are used from the processed SND_RDY packet and the matched slot. The size of the new transfer will be calculated by using the minimum of the size in the SND_RDY and in the slot.

The Outcommand Manager converts this packet to bulk DATA_MOV command for the COMM unit and request a notification when the RDMA Get is finished. Therefore, no special polling is required and the receive protocol sub-unit can handle the finished request as a simple command packet.

ACK ACK command packets are similar to SND_RDY packets regarding the actions which are invoked. It has no matching element, but directly informs the receiver about the which it is referring to. The receive protocol sub-unit has to check if the slot

was not already started. After retrieving the remote peer, the unit also has to check if no other slot has the same peer and sender slot id. A failure will drop the packet and no RDMA Get is started. Otherwise a new DMA_GET packet is generated as described in Section 5.3.2.

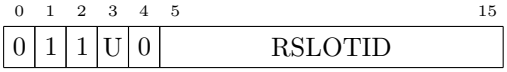


Figure 5.16.: DMA_FIN packet format

DMA_FIN A DMA_FIN packet is generated by the IN-Filter whenever a bulk DMA_MOV with notification was finished by the COMM unit. The sender has also to be informed to finish the transfer using a FIN packet from the receive protocol unit. The slot id from the DMA_FIN packet is used to retrieve the sender slot id from the preposted receive queue. Both information are enough to allow the sender to detect important mismatches in his preposted send queue when he receives the FIN packet.

FIN_ACK The answer of the remote send protocol sub-unit for a FIN packet is a FIN_ACK packet. The preposted receive queue has to be informed that the slot is no longer needed and can be reused for new transfers. No new packets are generated as reaction to this packet.

The schedule offset which was stored in the preposted receive queue slot has to be sent to the dependency resolver. All adjacent operations with a dependency counter of zero will automatically be started.

5.3.3. Incoming Non-synchronous Packets

The Non-synchronous InQueue can only receive information about finished RDMA Get operations in form of a DIRECT_DMA_GET packet. This packet has the same structure as a DMA_FIN packet as shown in Figure 5.16, but with the identifying bits `110X1`. The non-synchronous protocol sub-unit has to clear its internal state for the slot id which was stored in the packet.

The schedule offset which was stored in the preposted receive queue slot has to be sent to the dependency resolver. All adjacent operations with a dependency counter of zero will automatically be started.

5.3.4. Presorting the Active Queue

As explained Section 5.3, the active queue is not accessed directly by the protocol sub-units. Instead specialized queues are attached to the sub-units. Those queues only contain addresses to operations which can be processed by them. A unit called sorter is the filter which has to take the new elements from the active queue and identify the correct target queue.

The sorter can be implemented by using only two registers, some combinatorial logic and a control unit. One register is used to store the address from the active queue before it is written to the correct target queue. It is also used to read the type information from the address in the operation argument memory. The second register is used to save the extracted type information before a combinational logic selects the queue which is used to store the address from the first register.

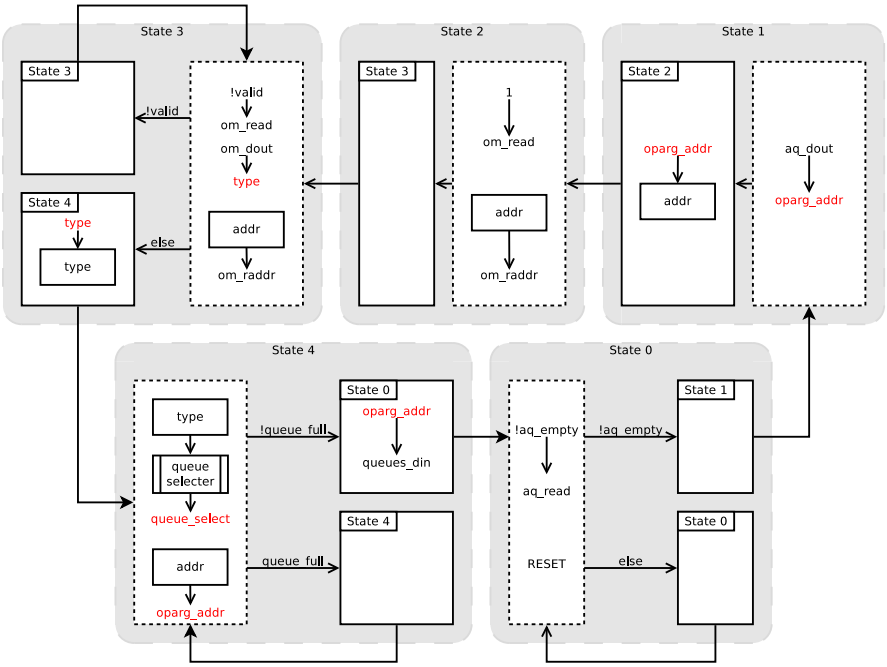


Figure 5.17.: Finite state machine for the Sorter

Figure 5.17 shows the state machine which controls these two registers and the memory attached to them. The unit waits in cycle zero until the active queue is not empty anymore. In this situation it will directly start a read of one element. This read element is the address which will be stored for later usage in register address. It is not directly send to the operation argument memory to reduce the critical path. This read is started in state two and repeated in state three until the read was successful

and the result is stored in the type register. A read could be unsuccessful when arbiter logic between the operation argument memory and sorter decided that another unit was allowed to access the memory. After a finished read the unit decodes the type and select the target queue. It will wait in state four until the target queue is not full anymore. The write signal will not be sent to the queue until not at least one entry is available to store the address. The unit will wait again in state zero for a new element in the active queue.

The sorter is capable of processing a new item from the active queue every five cycles when it is allowed to read directly from the operation memory, the target queues are not full and the active queue does not get empty.

5.3.5. Arbitration Units

The task of the different arbitration units is to support multiple attached units to access a component which can only handle one attached unit. Such a unit can have a simple interface like a write or read port of a simple dualport memory, but also a complex, command-based interface like the COMM unit.

The complex interfaces use specialized units called IN-Filter and Outcommand Manager that cannot be reused for other purposes. All other arbitration units are only variations of the same design. They are necessary to control the read access to the operation argument memory and notification of the dependency resolver.

The arbiter for the dependency resolver has four different input ports. Units which connect to these ports are the three protocol units and the unit which sends the initial finish information for address zero. Each port has a 16-bit wide address input and a signal that informs the unit that the address is valid. Each port also has an output signal that informs the unit whether the request to mark an operation as finished was successful in the last clock cycle. The dependency resolver unit cannot start to finish an operation when it is currently processing another one. Therefore, the connection to the dependency resolver does not only include the output for the address and the valid signal for the address, but also the busy signal input that the dependency resolver uses to inform the other units that it is currently not able to accept new finished operations.

The arbiter has to decide depending on the incoming signals which address is forwarded and whether the dependency resolver should get the signal that an operation was finished. The address which is forwarded gets chosen by a small prioritization unit that is directly connected to a multiplexer that connects the chosen input address

port to the address port of the dependency resolver, but the valid signal will only be sent to the dependency resolver when it is not busy. The decision about the winning unit will be saved in a register so that it can be used to inform the connected units in the next cycle. There is no winner when either no unit wanted to send a finished operation to the dependency resolver or when the dependency resolver is busy.

The input address is not chosen by combinatorial logic, but by a small prioritization unit that tries to prioritize all input ports in a round robin fashion. A register is used that can store four different states that define the different rounds. The priorities are equally distributed between the ports. There is no round were one port has two different priorities and no port has the same priority in two or more rounds. The prioritization unit chooses the port with a valid address that has the highest priority.

The arbiter that selects which of the three protocol units is allowed to read form the operation memory works using the same principles. The only difference is the busy signal. The memory has no busy signal, but a valid signal was introduced that informs the reading unit whether the read was successful. Therefore, the arbiter selects a port that wants to read in each round and sends the address from the selected port to the memory. The selected port is always saved in a register for the next clock cycle. The valid signal will only be one for the reading unit when the memory also signaled that the read was successful.

This extra valid signal from the memory comes from a simpler arbiter that manages the read access to the operation argument memory between the sorter unit and the arbiter which manages the reads from the three protocol units. It works similar to the previous arbiter, but has no round based prioritization unit. The sorter unit always has the highest priority to be able to fill the queues for the protocol specific units.

No extra cycles are added by the arbitration units, but other units which want to access the target component may have to repeat their request until it was successful.

5.3.6. IN-Filter

The COMM unit explained in Section 5.2 provides only a single data interface to the GOAL unit. Data received by the GOAL unit has to be analyzed and send to the specialized sub-protocol units which can interpret the content of a message and start actions corresponding to the protocol. Section 5.3.1 explains that each sub-protocol unit has a queue for incoming messages that is large enough to store the largest

packets for all slots that can be started at the same time to prevent deadlocks between two nodes.

The IN-Filter is the unit responsible for receiving the packets from the COMM unit. It is connected to the command interface to receive notifications and to the data interface to receive non-bulk transfers. Packets from the data interface are directly moved to the corresponding queue, but the notification is decoded to extract the user defined bits and to generate a new packet. The bit width of the bus to the COMM unit is 64 bit, but all other units try to use smaller buses to reduce the routing complexity. Therefore, the IN-Filter reads 64 bit words from the COMM unit and stores it in 32 bit queues.

The current implementation of the matching unit described was arbitrarily dimensioned to store 256 entries. The incoming queue for the send protocol, receive protocol and the non-synchronous protocol has to be calculated separately to store the different large packets.

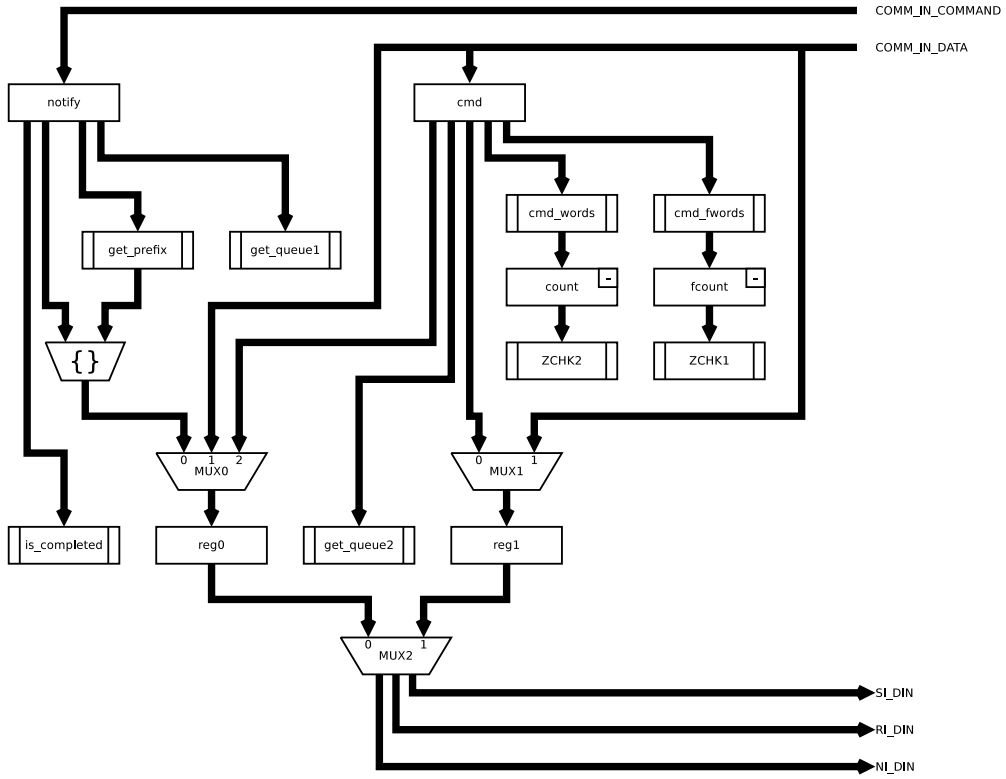


Figure 5.18.: Data Flow of the IN-Filter and connections to external resources

The send incoming queue can only receive two different types of commands. A RCV_RDY packet needs two 32-bit words and FIN only one 32-bit word that have

to be stored in the queue. A single 18 kib BRAM is enough to store the necessary 16384 bit. The non-synchronous incoming queue only has to store the slotid of the finished transfer which does not need more than a single 32 bit word. It is small enough to only use a 18 kib BRAM for this queue. In comparison, the receive protocol has more complex packets to process. A SND_RDY needs six 32-bit words and an ACK packet five 32-bit words. The simplest commands DMA_FIN and a FIN_ACK only need a single 32 bit word. Therefore, two 36 kib BRAM can be used to store the 49152 bit necessary for 256 SND_RDY commands in the receive incoming queue.

The IN-Filter is responsible for reading from the COMM interface and preparing the data for the queues. Figure 5.18 shows all components which are used in the IN-Filter. Signals from or to the COMM unit and the queues are not shown. Also the control unit is not shown to reduce the complexity. The IN-Filter can get data either from the command or data interface. A 64 bit word from the command interface is only processed as notification for a finished transfer and therefore only analyzed by combinatorial logic and then sent to the target queue. The 64-bit words from the data interface are parts of command message from other schedule interpreter units. There first bits have to be analyzed to find the length of a command packet and then all following 64-bit words have to be split into two 32-bit words before they can be sent to the target queue. The two interfaces have each a signal which has to be used to inform the comm unit that the schedule interpreter is busy. Therefore, it can happen that the registers notify and cmd are written in the first step, but the control unit will process first the notification before it processes the data. This should prevent that a single command packet and a notification message are interleaved in the same queue.

Figure 5.19 shows the state machine used to process the incoming data. State zero is used to analyze the internal state and the COMM unit valid signals. It checks whether a command packet header or a notification word was already stored. In that case it will not be overridden before it was not processed and the IN-Filter will signal a busy state using comm_in_stop and cmd_in_stop. An unused command or notification register will always be written when the COMM unit prefers to process DMA notification packets and ignores command packet headers until no new notifications are available. The *_in_stop signals are always one for all states but state one and six to prevent that the COMM unit sends more data that we cannot store.

The IN-Filter goes through state one and two to analyze the incoming notifications. The second bit of the notification bit is used to decide whether this notification is from a transfer started by the non-synchronous protocol unit or the receive protocol unit. This information is used to decide which queue is selected to store the packet and if the internal format for a DMA_FIN or a DIRECT_DMA_FIN is used. The created packet is stored in register reg0, the notification register is marked as empty

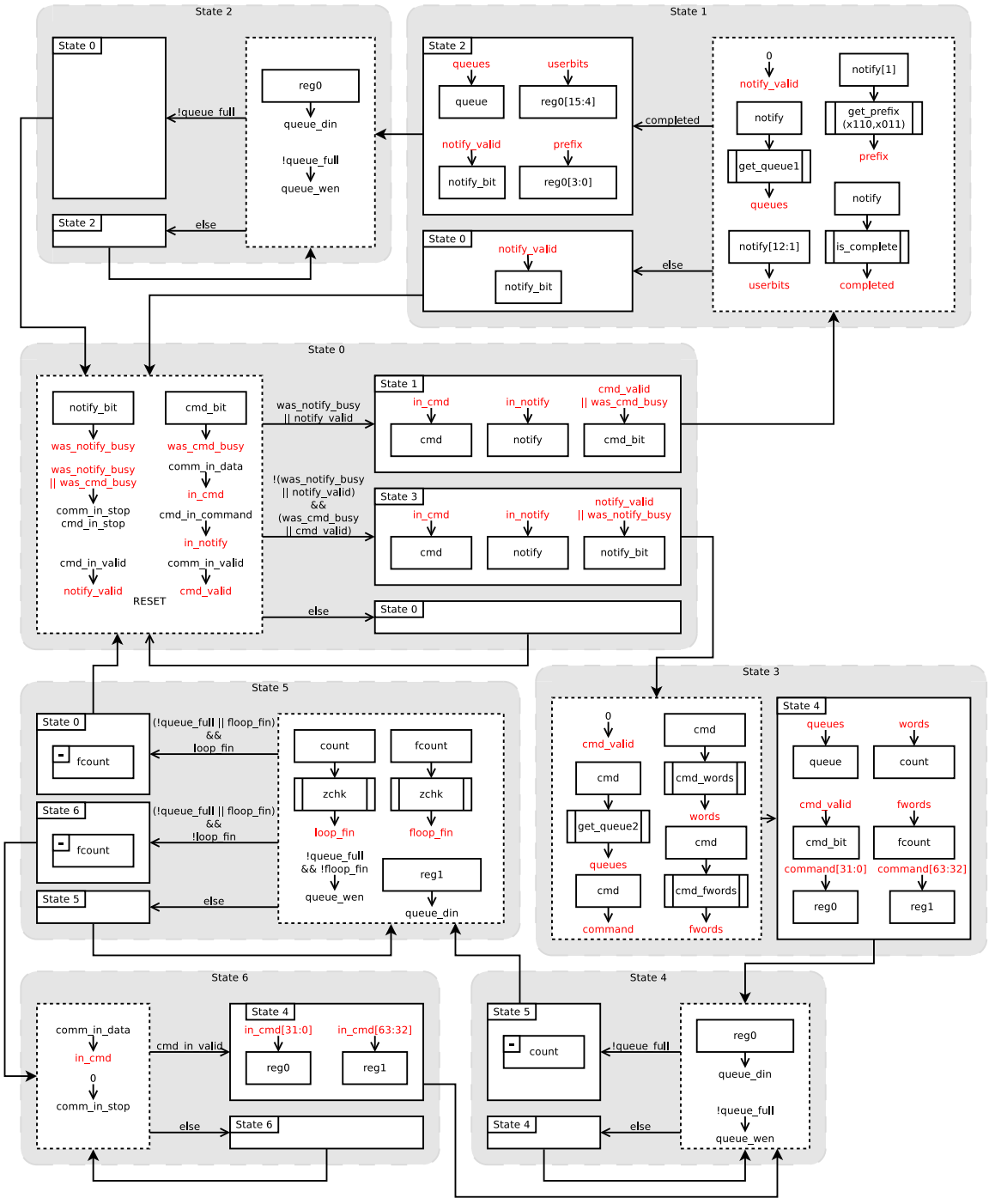


Figure 5.19.: Finite state machine for the IN-Filter

and the selected queue is stored in the register queue. It can happen that the received packet is not a notification but additional information that is not used by our protocol units. It drops this information and returns to state zero. State two waits until the target queue is not full before it appends the packet and returns to state zero.

The states three to six are responsible for reading the command packets and to put everything in the target queues. State three analyzes the headers of the command stored in the register, splits it into two 32-bit words and stores them in register reg0 and reg1. The command register can therefore be marked as empty using the cmd_bit register. The header bits are enough to calculate the length of the complete packet. The register count stores the amount of 64 bit words we read and fcount the amount of 64 bit words that are not only store 32 bit information. An example for a packet that has a smaller fcount than count is SND_RDY which is only 160 bits long and therefore has a count of 3, but an fcount of 2. The target queue is also selected using the command header and stored in the register queue. State four now waits until the selected target queue is not full anymore before it appends the first part of the packet and decreases count. State five does the same for the second part of the command packet, but only when fcount is not zero. It decreases fcount after a successful push operation and checks whether count is zero to decide if it can return to state zero. Otherwise it goes to state six where it tries to read the next part of the command packet directly into register reg0 and reg1. It has to set the comm_in_stop to zero to allow the COMM data interface to send data to the IN-Filter. Therefore, the data interface sends the IN-Filter a new 64-bit word when the signal cmd_in_valid is one. After a finished read, the IN-Filter goes back to state four to append the data to the target queues.

The IN-Filter needs at least three clock cycles to read a notification from the COMM unit command interface and append it to the target queue. The time it needs to write a command packet depends on the size of this packet. The first 32-bit word can be written after 3 cycles and the second 32 bit after an additional cycle. The third 32-bit need at least two more cycles. Therefore, the nth ($n \geq 1$) 32-bit word of a command packet takes at least $\lceil \frac{n}{2} \rceil + n + 1$ cycles to be appended to the target queue.

5.3.7. Outcommand Manager

The IN-Filter is responsible for receiving data from the shared COMM data and command interface. For send operations, the protocol sub-units have to use the Outcommand Manager. It prepares COMM unit commands to initiate the transfers and handles the send of data directly from the GOAL unit. It uses a similar approach with queues like the IN-Filter to prevent deadlocks between two GOAL units. The

elements in the queues are similar but not the same. Not all packets which have to be sent to another GOAL unit have the peer inside the command. Therefore, an extra 32-bit header is added before each packet by the sub-protocol units that stores the first 3 bits of the command, the local slotid and the peer id. This header is converted to the actual COMM unit command and removed from the packet payload.

The send outqueue can contain three different types of commands. The SND_RDY packet needs seven 32-bit words, an ACK needs six 32-bit words and a FIN_ACK needs two 32-bit words to be stored in the queue. Two 36 kib BRAM are enough to store the necessary 57344 bits for 256 SND_RDY commands with header. The non-synchronous queue can only contain DIRECT_DMA_GET request. These do not need an extra header to identify the peer because the first 32-bit of a DIRECT_DMA_GET request are already the header. 256 DIRECT_DMA_GETs use 57344 bit which can be stored in two 36 kib BRAM. The receive outqueue can contain RCV_RDY packets that need two 32-bit words, FIN packets that needs one 32-bit word and DMA_GET requests that need seven 32-bit packets. Therefore, the receive outqueue requires the same amount of BRAM as the non-synchronous outqueue.

The Outcommand Manager reads from the protocol specific output queues and sends the data to the COMM unit interfaces. Figure 5.20 shows all internal components that are used by the Outcommand Manager without the control unit and the signal wires of queues and the COMM interface. It selects one of the queues similar to the dependency resolver arbiter or the operation argument arbiter described in Section 5.3.5. The first 32-bit word is read and the unit decides depending on that information whether it creates a request to send a command packet to another GOAL unit or to transfer memory between a remote machine and the local host. A request for a command packet can be constructed using the information from the 32-bit header and actual data can be transferred to the COMM unit data interface whenever the COMM unit requests it and the Outcommand manager was able to read form the queue. But a request to transfer between the host memory requires the complete DMA_GET or DIRECT_DMA_GET packet and the COMM unit command interface also wants to get all 64-bit words of a command without any wait cycles. Therefore, the complete request is is buffered in a small queue called get_buf that is build using enough registers to store the complete request.

Figure 5.21 shows the finite state machine of the Outcommand Manager without the states to create a request to start a transfer between memory. State zero waits for a queue to become non-empty to read the first 32-bit word. A round robin like queue selector is used to choose one queue when multiple queues are non-empty. The selected queue is saved in a control unit register and used to provide the signal queue_empty, to send the signal queue_ren to the right queue and get the queue_dout from the correct queue using mux2. State one saves the first 32-bit in the register prefix to allow

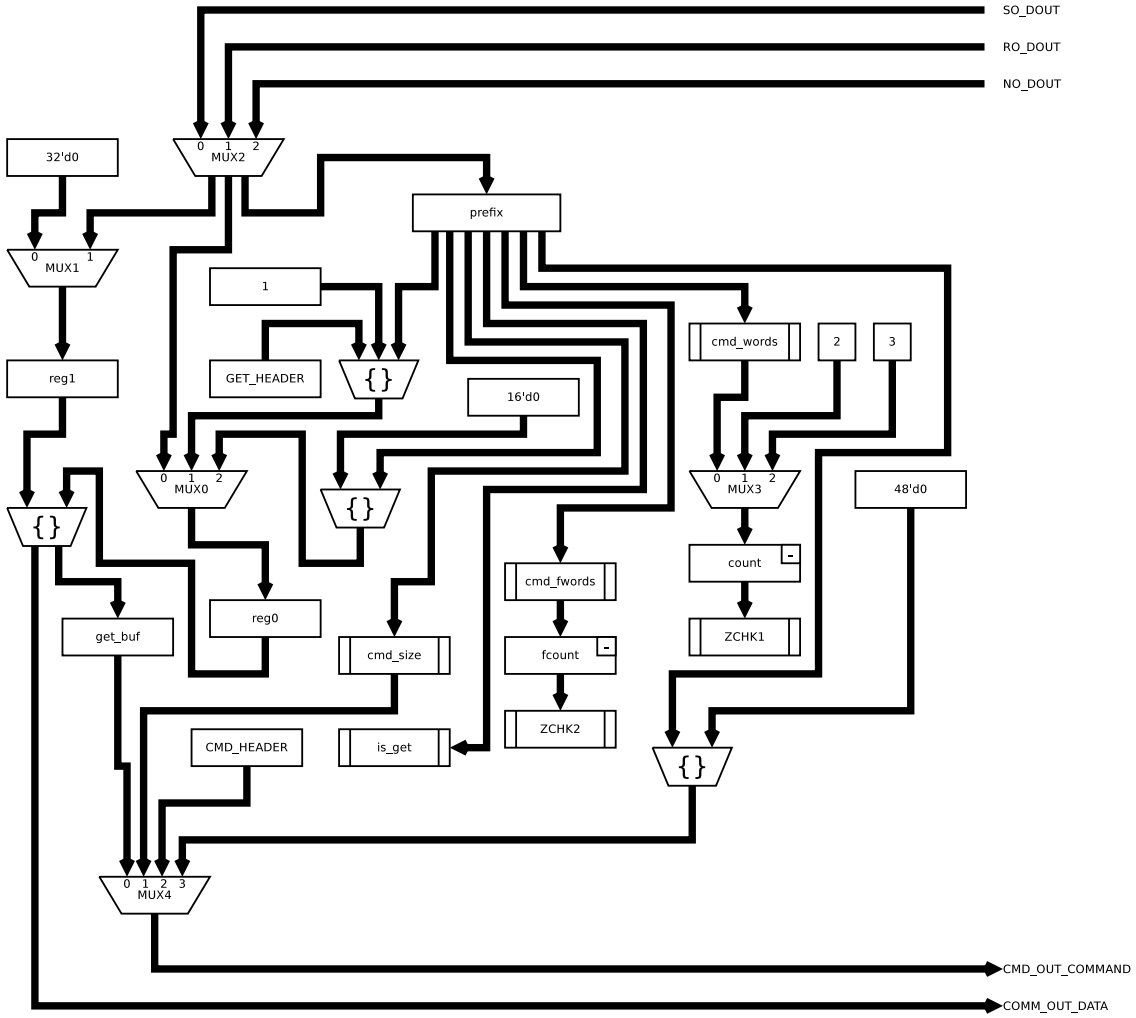


Figure 5.20.: Data Flow of the Outcommand Manager and connections to external resources

later states to create command that is sent to the COMM unit command interface. State three can now analyze the first three bits using combinatorial logic to detect a DIRECT_DMA_GET or a DMA_GET. State 11 is the next state in case that the header of a command packet was read. It has to wait until the non-bulk signal of the COMM command interface is not one to send a predefined bitstring that encodes the request for a non-bulk transfer without size and peer. The size is sent to the COMM cmd interface in state 12 and is determined using the first three bits of the header and a small combinatorial logic. This is also done to calculate the amount of full 64-bit words of payload and not full 64-bit words as described in Section 5.3.6. The command is finished in state 13 by sending the extracted peer id. It also tries to start

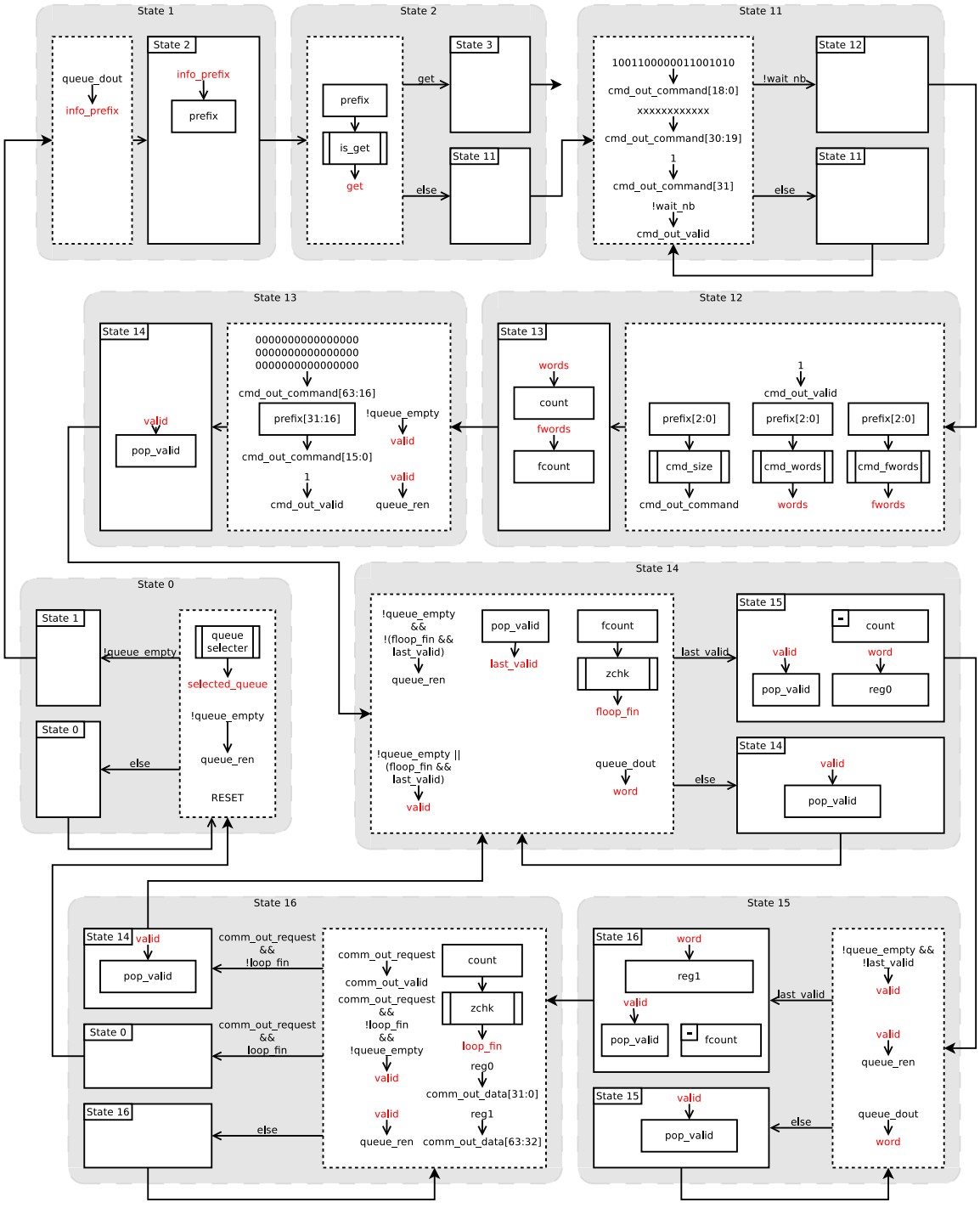


Figure 5.21.: Finite State Machine of the Outcommand Manager (command packets)

a read from the queue in case that it is not empty and stores in pop_valid whether it can use the output from queue_dout in the next cycle. Otherwise it will wait in state 14 until the queue is not empty anymore, the result was written in reg0 and the count was decreased. State 14 will request a the next item from the queue in case that it is not empty and not all 32-bit words from the command packet were already read. In case that all packets were read, pop_valid still will be set to one, but no read request will be made. It is not important that undefined data is send in this situation because the remote IN-Filter will drop it even when the COMM unit will transfer it. State 15 will either take the result from the last read request and save it in reg1 or wait until the queue is not empty anymore before it saves the result in reg1 and decreases fcount. State 16 concatenates reg0 and reg1 to a 64-bit word and sends it to the COMM unit data interface after the signal comm_out_request is one. It returns to state zero in case that count is zero or continues to read another 32-bit word from the queue and goes to state 14.

In case it detected a DIRECT_DMA_GET or a DMA_GET in state two, it will continue with state three to translate the internal packet format to a memory transfer request as shown in Figure 5.22. State three starts by writing the bitstring for this command and the local slotid into the register reg0 and reg1. These registers are directly connected to the queue get_buf and will be used in the next states to append more data to it. The count register is also initialized for a loop to read four additional 32-bit words for the size of the transfer and the local address. The outcommand manager requests a read from the selected protocol specific queue when the queue is not empty like in all states which are followed by a state that waits for the data from this queue. Otherwise it waits in state four until it could successfully save the output of the queue in register reg0. The old value of reg0 and reg1 is saved in the queue get_buf using the signal dma_cmdbuf_step and the count is decreased. State five also reads from the queue and saves the value to reg1, but does not decrease count or save the values of register reg0 and reg1 in get_buf. It will go again to state four in case that count did not reach zero or leaves the loop through state six. Here it saves the old values in get_buf and creates the remote rank identifier from the peer id stored in the prefix. State seven and eight are similar to state four and five. They try to read the remote address from the selected protocol queue, but do not use a loop for the two reads. All five 64-bit words for the COMM unit memory transfer are now stored in get_buf and in both registers reg0 and reg1. The Outcommand Manager has to wait in state nine until the COMM unit command interface does not have signal wait_b set to 1 to be able to send the request. It immediately starts with it and initializes count for a loop over the remaining four 64-bit words. State ten loops now four times and sends one value from get_buf to the command interface. It returns to state zero after the count reached zero.

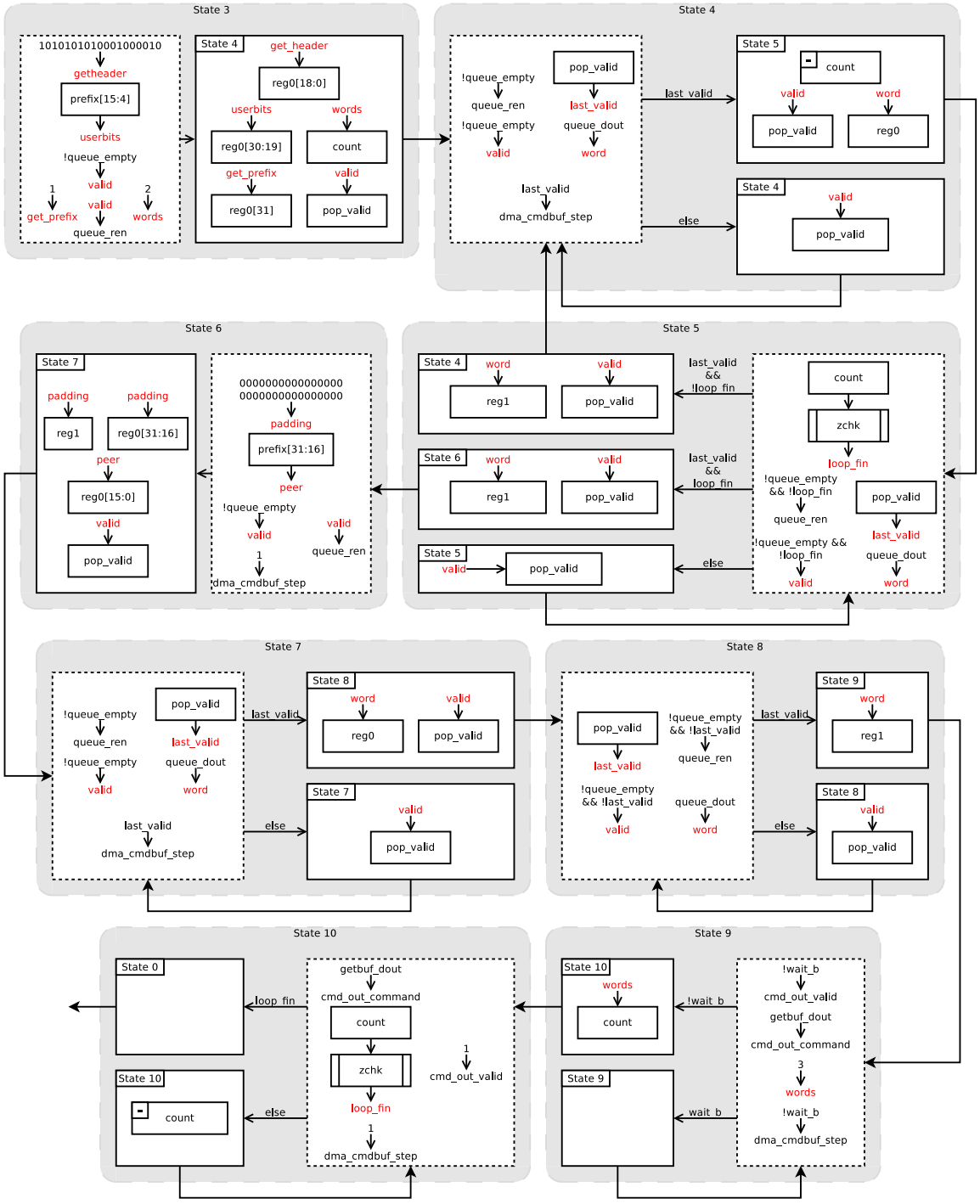


Figure 5.22.: Finite State Machine of the Outcommand Manager (memory transfer requests)

The Outcommand Manager needs at least 16 cycles to completely start a DIRECT_DMA_GET or DMA_GET. The time it needs to send a command packet depends on the size of it. For the first 64-bit, it needs at the minimum 9 cycles and for every additional started 64-bits 3 cycles more.

5.3.8. Non-synchronous Protocol

The Non-synchronous Protocol is the simplest sub-protocol unit in the current schedule interpreter implementation. It reads addresses to operation arguments from the presorted non-synchronous queue and uses them to start new operations. This can either be a DMA Get operations or no-ops. It sends new transfer requests to the Outcommand Manager through the outqueue and receives information about finished transfers from the IN-Filter through the inqueue. The position of the operation in the schedule memory is stored inside a simple slot memory that is explained in Section 4.5. The slotid is also send to the Outcommand Manager and returned as part of the DIRECT_DMA_FIN packet. Therefore, it can be used to identify the transfer after it was finished. This is sent to the dependency resolver to finish this operation and start now independent operations. The no-op operation type does not require additional actions and is directly send to the dependency resolver to mark it as finished. All components required for this including the connections to external units are shown in Figure 5.23. The control unit and signals which are only connected to the control unit like the output of IS_GET or ZCHK are not shown to reduce the complexity.

The register offset is used to store the address to the operation argument memory that was read from the presorted non-synchronous operation queue. It is also increased to read the complete data that are stored about the operation in the memory. This includes the type of the operation that is analyzed again to differentiate between a DMA Get and a no-op. This is done using combinatorial logic and used inside the control unit. Also the address of the operation in the schedule memory are stored in schedoff and is sent from there to the non-synchronous slot memory. It cannot only come from the operation argument memory, but also has to be read back after a DIRECT_DMA_FIN was received. In that situation and in case a no-op was detected, it will also send this address to the dependency resolver. The output of inqueue can only be a notification of finished transfer and can be stored directly in tmp_buf to be forwarded to the non-synchronous slot memory to retrieve the schedule address. All remaining parts of the operation argument memory are saved in the register cmd_word to be transferred to the Outcommand Manager. This upper and the lower 16-bit of this register can be written independently to create a 32-bit word for the Outcommand Manager from the 16-bit wide words of the argument memory.

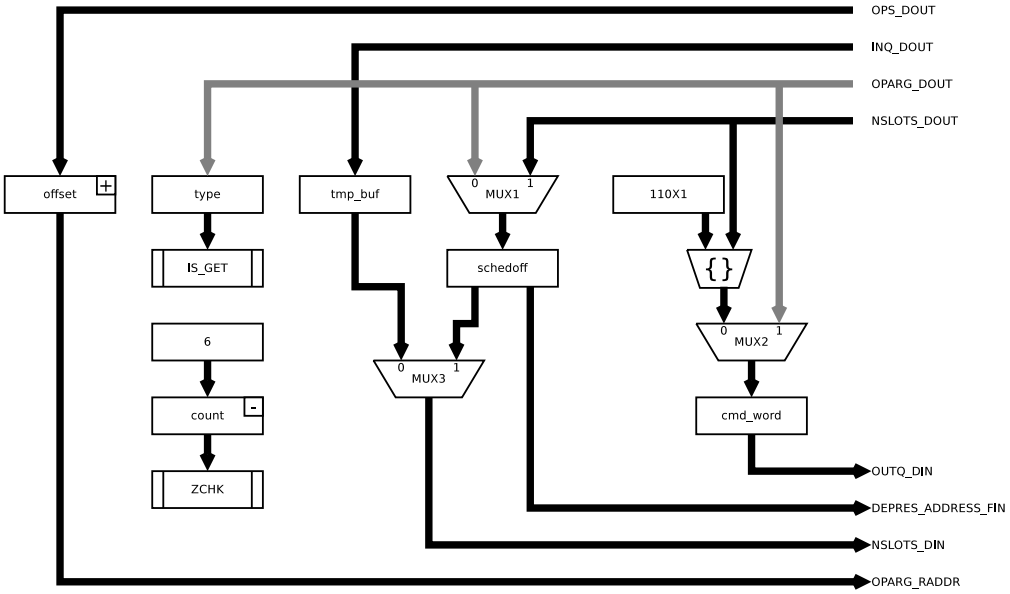


Figure 5.23.: Data Flow of the Non-synchronous Protocol

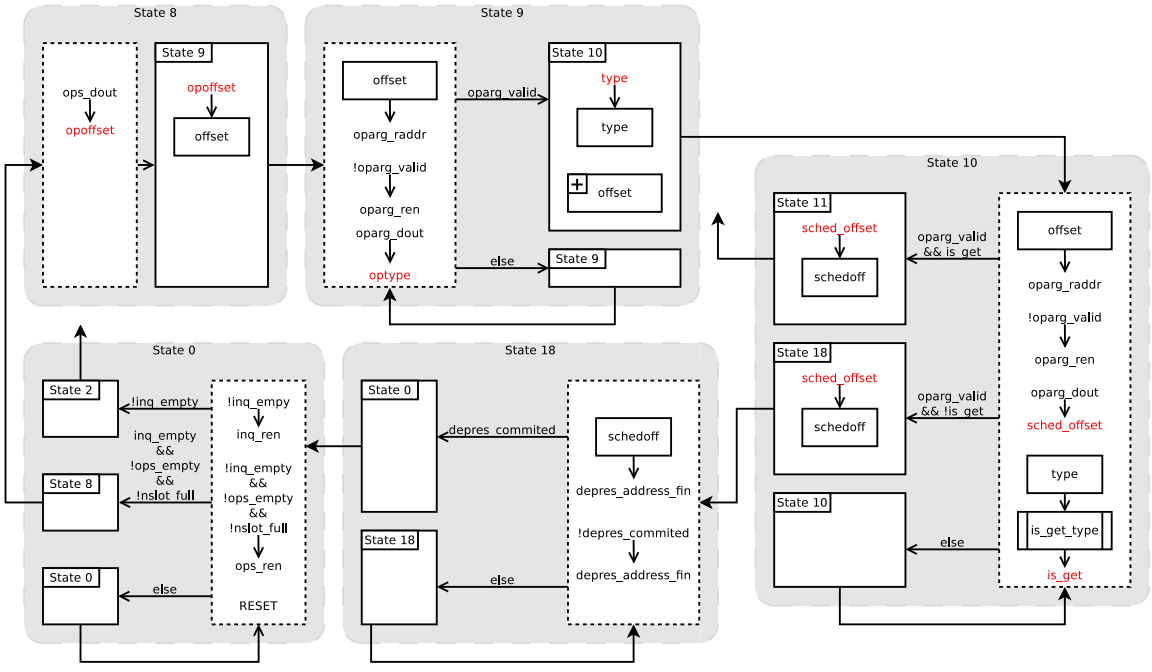


Figure 5.24.: Finite State Machine of the Non-synchronous Protocol (Starting of a no-op)

Figure 5.24 shows the states of the Non-synchronous protocol that are responsible for starting a new operation and all actions necessary if the operation was identified as no-op. In state zero, it tries to detect whether new DIRECT_DMA_FIN packets are in the inqueue and starts to read them. This should keep the number of used slots low because a FIN packet frees a slot. Otherwise it tries to read a new operation when the slot memory is not full. State eight saves the result of the read in offset and state nine uses it to read the type of the new operation and saves it in register type. The Non-synchronous protocol remains at least two cycles in state nine because the output of the operation argument memory can only be valid one cycle after the read was made. It can take even longer when another sub-protocol unit tries to access the memory. The register count will also be increased after an successful read to point to the schedule memory offset of the operation. State ten tries to read from this address and goes to state 18 when it detects a no-op and to state 11 when it detects a DMA Get operation. State 18 waits until the dependency resolver accepted the finished operation that this state sends every time it was not successful and returns to state zero after that.

The state machine to start a DMA Get is shown in Figure 5.25. The offset of the operation in schedule memory was already stored in schedoff by state 10. State 11 can now use this register to send an add request to the non-synchronous slot memory when it is not busy and move offset to the first byte of the Direct DMA Get specific argument. State 12 now has to wait until the non-synchronous slot id is valid before it saves it together with the header to the lower 16-bit of the register cmd_word. The current implementation of the non-synchronous slot add mechanism forces only a single additional wait cycle. State 12 can continue with reading the peer and saving it to the upper 16 bit of the cmd_word register. The complete 32-bit are written in state 14 to the outqueue unless it is full and count is initialized to write the remaining 6 32-bit words. The loop for this functionality is implemented using state 15, 16 and 17. State 15 and 16 read the content of the operation argument to the register cmd_word similar to state 12 and 13, but use the complete data and not only parts of it. The counter is reduced in state 16 after a successful read. State 17 is responsible for appending the register to the outqueue and returns to state zero after count reaches zero.

Finished transfers are directly detected in state zero when it starts to read from the inqueue. The local slot id is extracted from this DIRECT_DMA_FIN packet and stored in tmp_buf. State three can start a read when the non-synchronous slot memory is not busy to extract the schedule offset for the finished operation. State four saves it into sched_off after an extra wait cycle that is necessary for the data on nslots_dout to become valid. State five uses the offset to inform the dependency resolver and state six frees the slot. The Non-synchronous protocol unit can return to

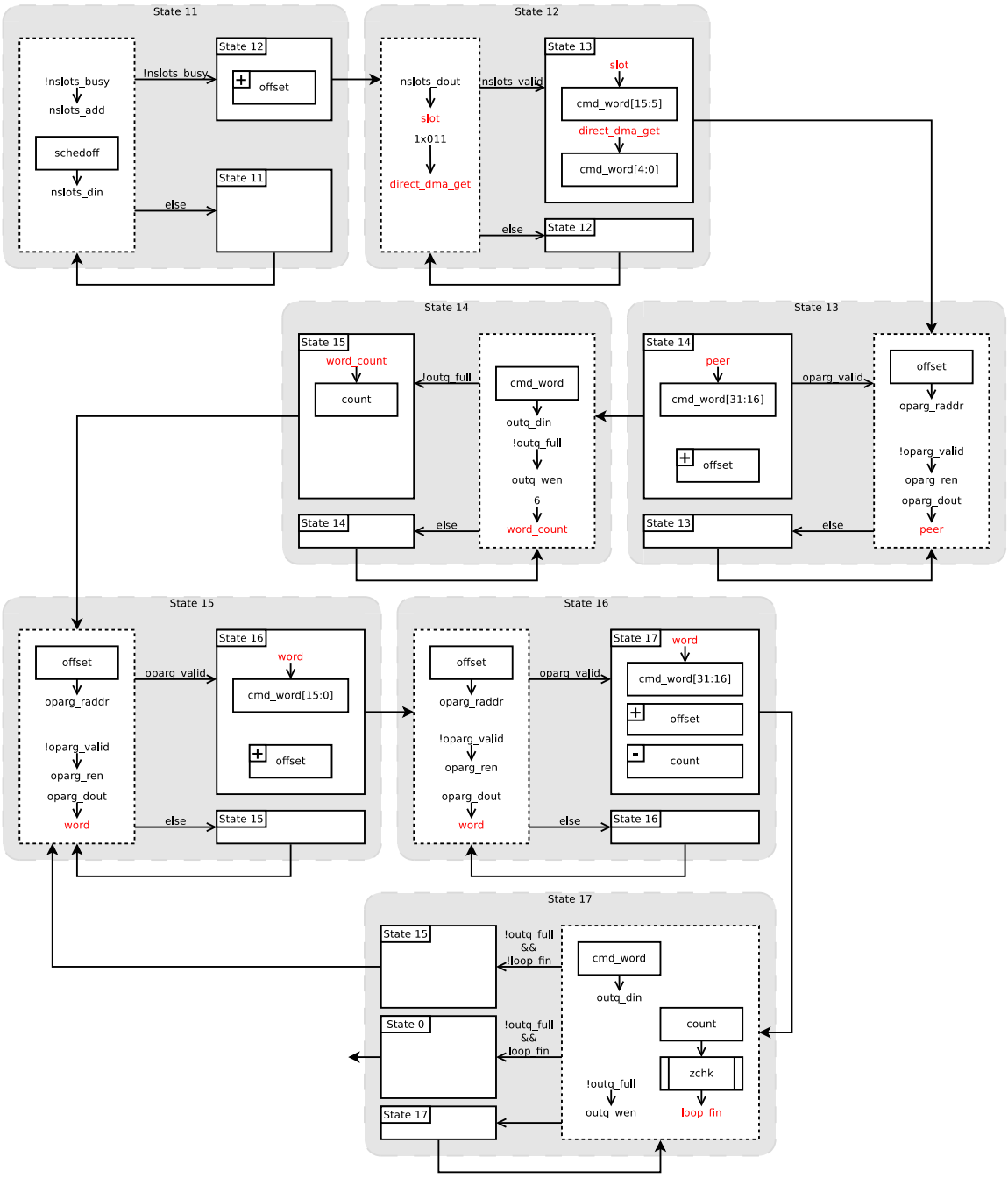


Figure 5.25.: Finite State Machine of the Non-synchronous Protocol (Starting of a Direct DMA Get)

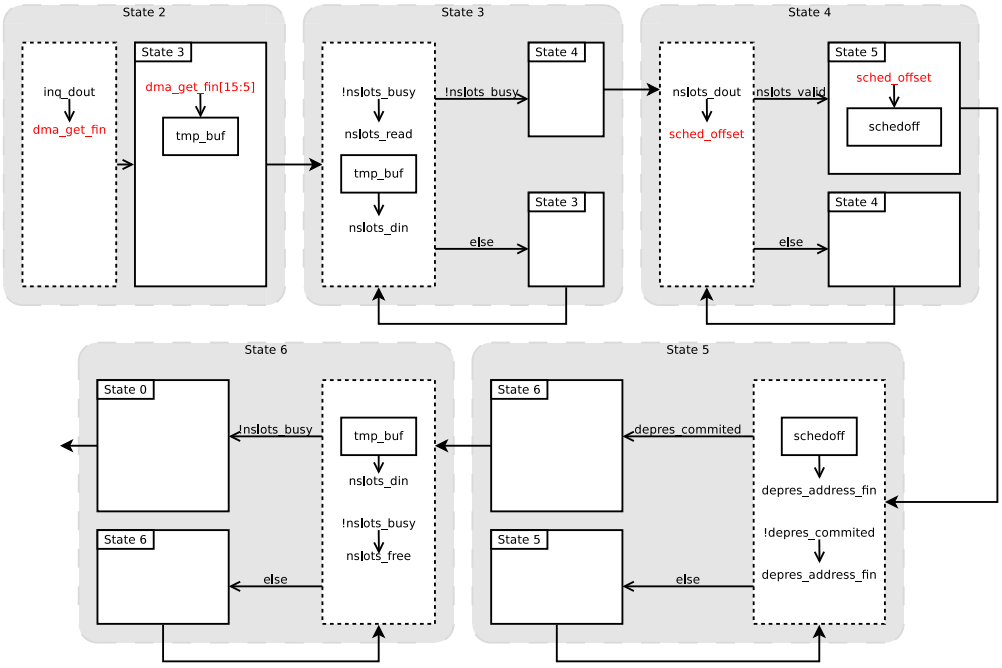


Figure 5.26.: Finite State Machine of the Non-synchronous Protocol (Processing of a Direct DMA FIN)

state zero after both finished and start process new incoming data from the different queues.

The Non-synchronous protocol unit needs at least 8 cycles to finish a no-op. It is not possible to tell how long a DMA Get takes without knowing the details of the network and the COMM unit, but it is possible to predict the time it takes to start a DMA Get and to finish the operation after receiving the DIRECT_DMA_FIN packed. The start of the DMA Get needs at least 12 cycles to write the first 32-bit word to the outqueue and 30 cycles to write the remaining 6 32-bit words. The appending of the last six words are overlapped with the Outcommand manager that tries to read from the queue at the same time to initiate the transfer using the COMM unit. The processing of the FIN packet takes at the minimum 8 cycles.

5.3.9. Send Protocol

The task of the Send Protocol unit is to implement the sender part of the synchronous protocol explained in Section 5.3.2. It fetches addresses to new operation arguments and starts them by sending a SND_RDY packet. The data of the ongoing transfers

are stored inside the preposted send queue which is an instantiation of the matching unit developed in Chapter 4. It receives command packets through the inqueue and processes them depending on the type. Not only the storage but also the matching capabilities of the preposted send queue are used to find unmatched slots with the same matching element or to find slots that need to be restarted.

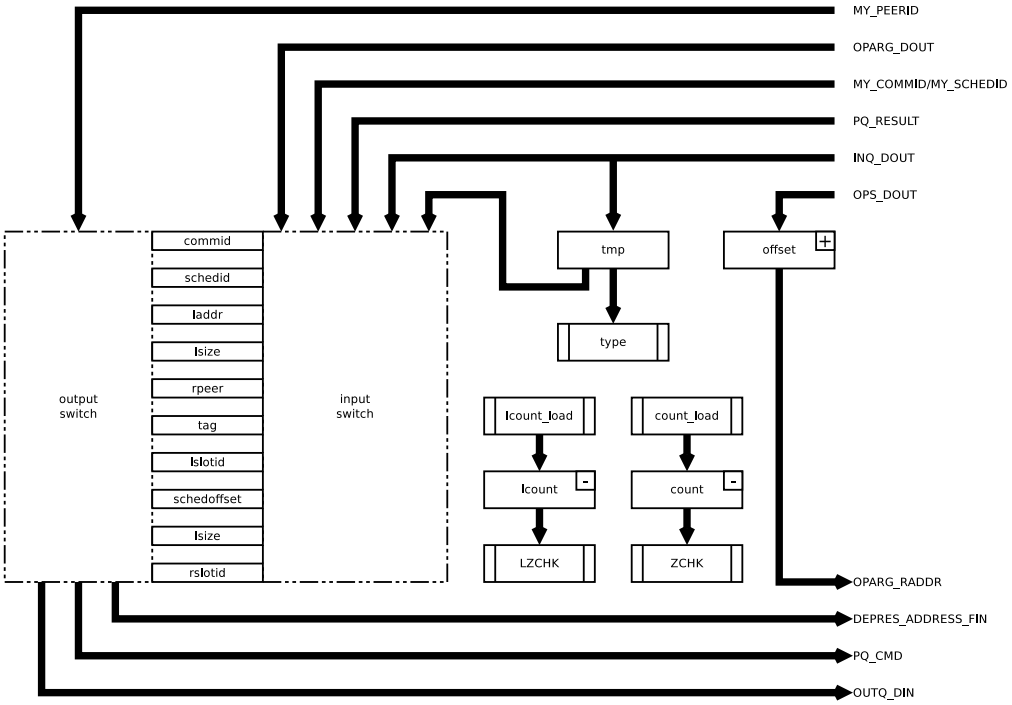


Figure 5.27.: Data Flow of the Send Protocol and connections to external resources

The components of the Send Protocol are depicted in Figure 5.27. The control unit and signals from and to the control unit are not shown to reduce the complexity. Also not all data paths are shown and instead are replaced by the boxes “input switch” and “output switch”. The input switch connects nearly all input signals to the the target registers between input switch and output switch. Only connections that are used in the state machine of the Sender Protocol are implemented to reduce the number of wires and multiplexers required in hardware. An example for such a register with missing connections is the schedoffset that stores the address of the current operation inside the schedule memory. It will only receive its content from the operation argument memory and the preposted send queue. It is never send to or received from another node and can therefore never come from the inqueue. Another special behavior is the splitting of input signals. For example the inqueue stores 32-bit words, but the registers rpeer and tag are only 16-bit. This 32-bit word is splitted in two 16-bit words and each part is sent to a different register. The output switch

has the same purpose, but connects the internal registers to the external components. Therefore, it can take two registers and append the resulting 32-bit word to the preposted send queue.

The internal registers between input and output switch contain temporary data used when processing a packet or starting an operation. Registers prefixed with an "l" contain data from the local unit and registers prefixed with an "r" data generated by the remote protocol unit. The tmp register is a 32-bit register that stores the header of a packet before the type of an operation was identified. This is necessary to be able to find the correct target registers one cycle after the first 32-bit word was read from the inqueue. The register offset receives the address to the operation argument and is used to specify the address while storing a send argument inside the preposted receive queue. The registers count and lcount are used as loop counter in various states. The control unit can send different signals which are decoded by the combinatorial logic lcount_load and count_load and can be used to initialize both registers. The Figure only contains the combinatorial logic to compare both register against zero to detect a finished loop, but the content of both registers is used by the control unit to decide which paths in the input and output switch have to be enabled and which register should receive a write enable signal.

Figure 5.28 shows a simplified overview of the Send Protocol state machine. The complete state machine can be found in Appendix A. The detailed diagrams contain registers with three dots and dashed rectangles next to the states. The text in those rectangles explain under which condition which register or signal is used. Usually the output signal depends on the count register, but it can also depend on lcount or both registers.

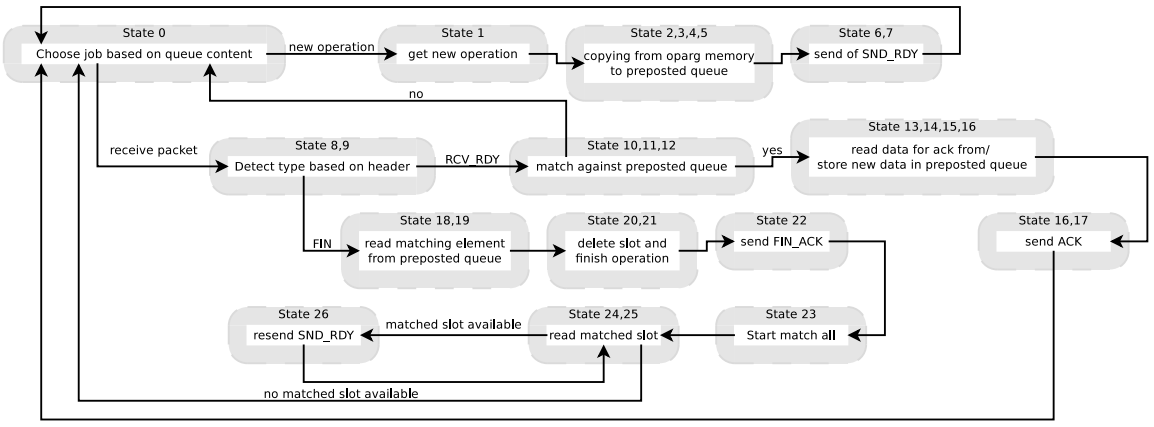


Figure 5.28.: Overview of the Send Protocol State Machine

The overview state machine only shows the main work for the Send Protocol unit and which states implement which task. Only the state changes between combined states are shown without internal wait cycles or loops. The basic concept is similar to the Non-synchronous protocol but uses a more complex packet handling. State zero tries to read new incoming data from the inqueue or starts new operations. Incoming data is prefered to respond to command packets as fast as possible and to keep the number of elements in the protocol queues small. The states one to five get the address of the operation argument, load it into the internal registers and insert them into the preposted send queue. The data from the internal registers and the local slot id are also used in state six and seven to prepare the SND_RDY packet and send it to the Outcommand Manager before the unit returns to state zero. Incoming packets are identified in state eight and nine by saving the first 32-bit in the register tmp and using the combinatorial logic type to decode the first three bits. In case that the received command packet is a RCV_RDY, the Sender Protocol unit reads the complete packet into the internal registers and starts a MATCH_ANY request using the matching element in states 10 to 12. It waits that the preposted send queue is not busy anymore and returns a local slot id. No returned slotid means that no matching element was found and the RCV_RDY has to be dropped. Otherwise additional information received through the RCV_RDY is stored in the preposted send queue. An ACK packet is created using the data stored in the matching slot and send to the Outcommand Manager by states 13 to 17.

A received FIN packet results in a read of the matching element of the finished slot in state 18 and 19. The states 20 and 21 can immediately delete the slot, send the address of the finished operation to the dependency resolver and inform the receiver through a FIN_ACK packet in state 22 about the received FIN packet. The saved matching element is used in state 23 to request a MATCH_ALL to find all slots that may have a different state in the Receive Protocol unit or were dropped by the receiver. The preposted send queue directly provides the data necessary to resend the SND_RDY packets by states 24 to 26. The Send Protocol unit returns to state zero after the preposted send queue is not busy anymore and no data can be found in the queue between both units.

5.3.10. Receive Protocol

The Receive Protocol unit is responsible for the receiver part of the synchronous protocol explained in Section 5.3.2. It fetches addresses to new operation arguments and starts them by sending a RCV_RDY packet. The data of the ongoing transfers are stored inside the preposted receive queue which is an instantiation of the matching unit developed in Chapter 4. It receives command packets through the inqueue and

processes them depending on the type. Its task is also to start memory transfers using matching operations from the preposted receive queue and the sender. It can detect different inconsistent states between both sides and can decide which transfer is started and which is postponed.

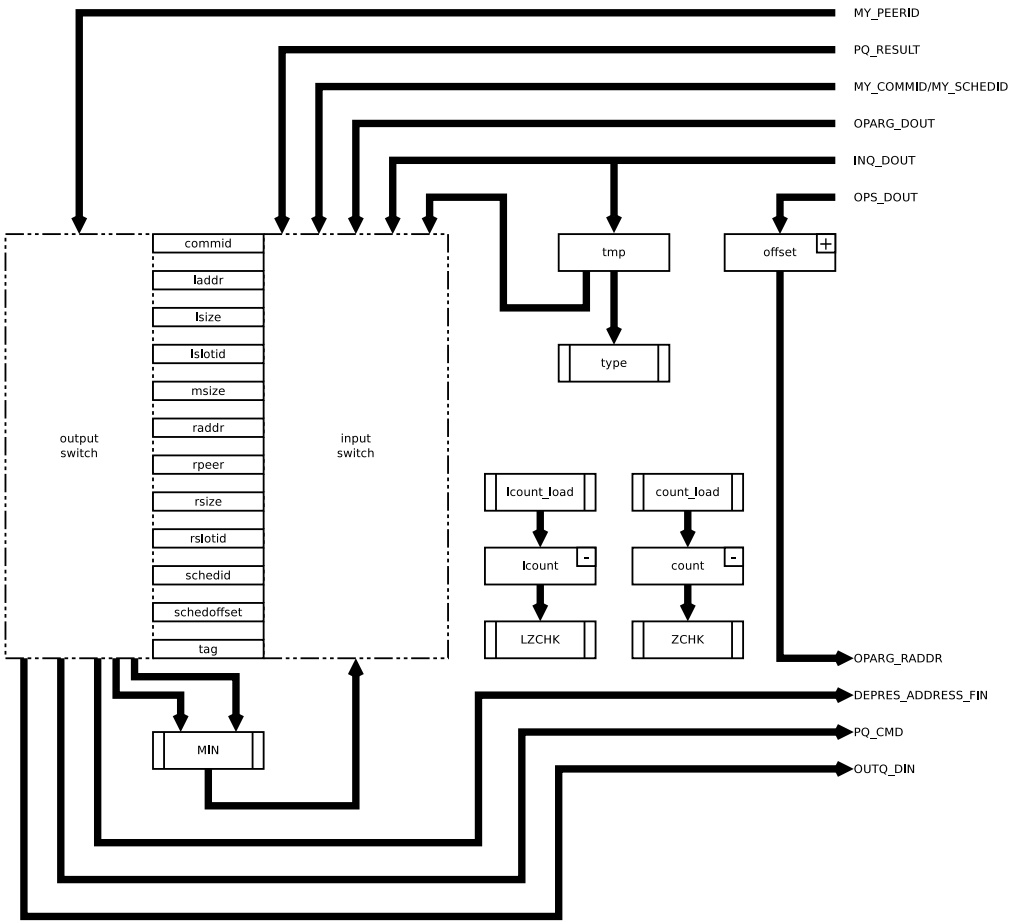


Figure 5.29.: Data Flow of the Receive Protocol and connections to external resources

The components of the Receive Protocol are depicted in Figure 5.29. The design works much like the Send Protocol unit, but provides some extra registers and combinatorial logic that is necessary to create a DMA_GET request. An example is the MIN component that calculates the minimum of the local and remote size. This is necessary as the sender and receiver can have different sizes stored inside their operation argument. Therefore, it is only save to start a memory transfer with the minimum of both sizes. All other components have the same purpose as in the Send Protocol but were slightly adjusted to parse and create the different packet types.

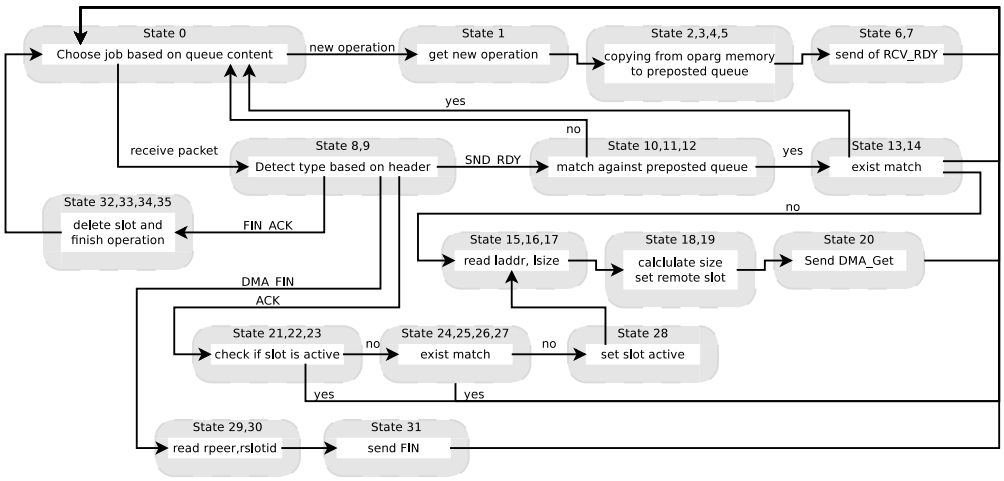


Figure 5.30.: Overview of the Receive Protocol State Machine

Figure 5.30 shows a simplified overview of the Receive Protocol state machine. The complete state machine can be found in Appendix B. The decision which queue is read in state zero works the same way it did in the Send Protocol unit. Also the starting of operations in states one to seven are similar to the start of operations in the Send Protocol, but with the difference that a RCV_RDY is sent. The Receive Protocol can detect other types of packets in state eight and nine that need to be processed differently. A SND_RDY packet has to be matched against the preposted receive queue in state 10, 11, and 12 to find a local slot that is not yet started and has the same matching element. An unmatched RCV_RDY is dropped the same way as an SND_RDY in the Send Protocol, but the unit must also check whether the remote slot in the SND_RDY is not already participating in another transfer. This is done by the EXIST matching of the preposted receive queue in state 13 and 14. After no slot was found, the unit can continue to create the DMA_GET request. It has to read the local address and size from the preposted queue in states 15 to 17. The remote slot has to be stored in the preposted queue to be able to check for a duplicated ACK or SND_RDY of an already started transfer. At the same time in state 18 and 19, the minimum of the local and the remote size has to be calculated to create the DMA_GET request from the internal registers in state 20.

An ACK is processed similarly, but does not need to find the corresponding local slot by comparing the matching element. It has only to be checked that this local slot is not already started before the unit has to search for conflicting transfers and and mark the active slot as active. That is why the initial part in states 21 to 28 is implemented separately, but the creation of the DMA_GET request in states 15 to 20 is shared between SND_RDY and ACK processing.

The IN-Filter announces finished transfers to the Receive Protocol unit through a DMA_FIN packet. It contains the local slot id which is used in state 29 and 30 to read the remote peer and the remote slotid. These are used in state 31 to send a FIN packet. The processing of a FIN_ACK in states 32 to 35 also uses the included local slot id to read the address of the operation in the schedule memory and uses this to inform the dependency resolver. The slot is freed after the dependency resolver accepted the address.

5.3.11. Local Operations on FPGA

GOAL defined not only network operation, but also user defined functions that can be executed by the main CPU. This concept was replaced during the development of ESPGOAL by local operations that provide a predefined set of arithmetic and logic operations as explained in Section 2.4.1. A similar approach can also be taken by a hardware implementation to also allow offload of reduce operations. An implementation of a localop unit could be done similarly to the protocol units. They must be able to receive data, start the calculation and send the result again the the COMM unit.

The received data is processed in the current implementation by the IN-Filter which parses the header of a packet and sends it to the protocol unit that can handle the packet type. Data which would have been requested from the host memory or another GOAL unit would not have such a header and therefore no identification bits which select the correct target queue. An additional information channel is the address send by the COMM unit to the IN-Filter. The Outcommand Manager can choose a address range to select different address for command packets or for local operations. This can also be used to select which type of arithmetic operation and which element size should be selected.

Different units capable of processing a specific arithmetic operation were implemented to check whether arithmetic operations can be implemented well on an FPGA. Floating point operations where implemented using the “Xilinx DS816 Floating-Point Operator v6.0” [Inc11a] and division of unsigned integers using the “Xilinx DS819 Divider Generator v4.0” [Inc11c]. All other operations were implemented using builtin Verilog statements. The IP core generators allow to change different parameters that influence the achievable clockrates of the implemented design. The main parameter is the latency of the unit. A target clockrate of at minimum 200 MHz was choosen to tune this parameter. Nearly all units are fully pipelined and therefore can start one operation each cycle. The units generated by the divider generator are not always

pipelined and only allow to start a operation every n cycles, e.g. a unit for 16-bit integer division can only start an operation every 4 cycles.

Operation	Bits	Latency	Throughput	Speed	
				ns	MHz
Integer AND/OR	32	0	1 in 1	1.052	950.570
Integer AND/OR	64	0	1 in 1	1.088	919.118
Integer XOR	32	0	1 in 1	1.102	907.441
Integer XOR	64	0	1 in 1	1.124	889.680
Integer Add	32	0	1 in 1	1.559	641.437
Integer Add	64	0	1 in 1	2.136	468.165
Integer Subtraction	32	0	1 in 1	1.559	641.437
Integer Subtraction	64	0	1 in 1	2.136	468.165
Integer Min/Max	32	0	1 in 1	1.456	686.812
Integer Min/Max	64	0	1 in 1	2.066	484.027
Integer Multiplication	8	0	1 in 1	2.104	475.285
Integer Multiplication	16	0	1 in 1	3.320	301.205
Integer Multiplication	32	2	1 in 1	3.735	267.738
Integer Multiplication	64	3	1 in 1	4.229	236.463
Integer Division	8	9	1 in 1	4.972	201.126
Integer Division	16	14	1 in 3	4.968	201.288
Integer Division	32	16	1 in 4	4.972	201.126
Integer Division	64	31	1 in 5	4.829	207.082
Floating-Point Add	32	5	1 in 1	3.810	262.467
Floating-Point Add	64	5	1 in 1	4.440	225.225
Floating-Point Subtraction	32	5	1 in 1	3.818	262.261
Floating-Point Subtraction	64	6	1 in 1	3.750	266.667
Floating-Point Multiplication	32	2	1 in 1	4.538	220.361
Floating-Point Multiplication	64	5	1 in 1	4.533	220.604
Floating-Point Division	64	29	1 in 1	4.853	206.058
Floating-Point Division	32	10	1 in 1	4.807	208.030
Floating-Point Min/Max	32	0	1 in 1	2.010	361.141
Floating-Point Min/Max	64	0	1 in 1	2.802	356.888

The results of these benchmarks can be used to estimate the amount of time it takes to finish n operations and compare it against a program running on the host cpu. Figure 5.31 shows a comparison between an AMD Opteron 285 using a C++ loop that multiplies two different buffers with 32-bit floating point numbers and the above numbers without additional overhead. Therefore, it is assumed that the FPGA already has the operands in a local buffer and can read both operands in one cycle.

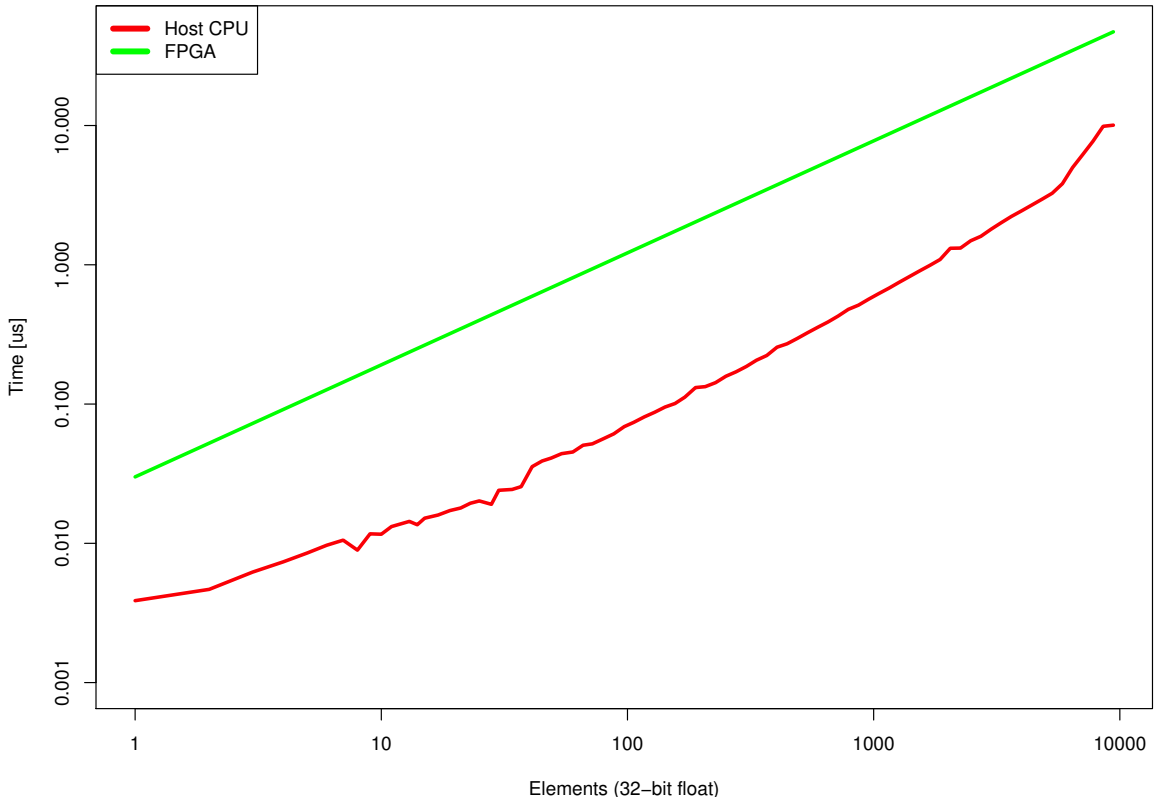


Figure 5.31.: Speed comparison of the execution of C++ program performing floating point adds and the expected performance of an FPGA IP Core

The benchmark shows that the host CPU running at a higher clock rate is faster for the tested number of elements. An extra localop unit can still reduce the amount of overhead introduced by the copy operation to the host memory and the interrupt latency (about $1\mu s$ on a modern Linux system) to start the calculation on the host CPU.

6. Evaluation

6.1. Performance Analysis

Each part of the design was synthesized and implemented by Xilinx ISE 13.2 for a Virtex6 (XC6VLX75T-3ff784). Each module was analyzed using post-place and route static timing.

A speed test module was used for each component to simulate the synchronous producers and consumers to get realistic results for the timing of the longest critical path. It was designed to behave similar to the test modules used by Xilinx [Inc11a] to reduce the effects caused by the input and output pad mapping.

Each input wire of the unit under test was connected to a synchronous register array using the same clock as the unit. The register array was connected to another register array using a separate clock and getting its input from input pads. The output of the unit under test was also connected to a synchronous register array using the same clock as the unit. This register array was attached to another synchronous register array using the same clock as the outer input register array. The outer output register array was attached to the output pads. All inner components got their clock signal from two clock wires connected over a `IBUFGDS` to provide a single clock signal. The input clock `TNM_NET` had a timing constraint of 1ns. The registers in the testing module got a KEEP constraint to prevent their removal during the optimization process or changing their position by the register balancing algorithm.

The synthesis optimization goal was set to "speed" with an high effort level and it was allowed to remove the hierarchy during that process. Map, place and route were configured to ignore user timing constraints and to use a timing model for performance evaluation. The combinatorial optimization and global optimization for the map process was activated. Packing of registers into IOBs was deactivated to ensure that only the internal design with the extra input and output registers are used for the timing evaluation.

	FlipFlop	LUTs			RAM		MHz
		Logic	Memory	route	B18	B36	
Active Queue	37	57	0	2	0	2	404
Compare	120	37	0	23	0	0	677
Dep. Resolver	121	157	0	7	0	0	483
DepRes_Arbiter	7	89	0	0	0	0	556
DMA Queue 512	27	31	0	2	1	0	403
DMA Queue 2k	34	71	0	2	0	2	386
Freelist Queue	24	28	0	2	1	0	466
Freelist	33	49	0	3	1	0	440
InFilter	196	186	0	14	0	0	422
Input Consumer	91	103	0	0	0	0	580
Matcher	459	703	3	17	5	1	275
Mem. Arbiter	6	71	0	0	0	0	1104
Nons. Protocol	124	125	0	7	0	0	487
Nons. Slots	60	85	0	3	0	0	421
OpArgs Arbiter	2	18	0	0	0	0	805
OpArgs Memory	4	80	0	2	0	16	366
OutCMD Manager	208	264	64	0	0	0	353
Output Generator	149	238	0	3	1	0	340
Presorted Queue	30	34	0	2	1	0	436
Recv. Protocol	503	701	0	23	0	0	407
Schedule Memory	2	20	0	0	0	4	379
Send Protocol	304	628	0	0	0	0	369
Slot Memory	0	0	0	0	0	1	406
Slot Manager	101	138	0	7	2	0	241
Sorter	35	45	0	0	0	0	540
Usedlist	73	75	0	3	1	0	369
Schedule Interpreter	2803	3762	79	95	17	32	194

The static timing identified during the performance evaluation mode can be used for rough comparison of the components. We suspect that the Xilinx tools often stop the optimization process before the global optimum was found, because slight changes in the Verilog code, such as the introduction of an unconnected wire, sometimes lead to better static timing results. A user constraint clock timing can force a higher map, place and route effort and therefore better timing results in case that the user specifies a possible timing higher than the previously evaluated one. The complete schedule interpreter unit was analyzed by testing different user constraints. The values between 194 MHz and 388 MHz were bisected until the highest possible MHz constraint was found that was still achievable. The lower limit was chosen to be

the initial performance evaluation result and the upper limit to a value which was suspected to be impossible to achieve.

The highest possible constraint for the complete schedule interpreter design was 212 MHz. The critical paths which limit the possible clock timing are between BRAM and the first register in the unit which either has a read or write port attached to it. The delay which limits the clock timing is a combination from both net and logic delay. The actual critical path depends on the optimizer settings and especially on the clock timing constraint.

In general, the clock-to-out delay of the memory together with the logic to select the correct BRAM output in multi-BRAM memory blocks are a main reasons for the logic delay. But the critical path for the implementation running at 212 MHz has 45% logic delay and 55% net delay. The amount of net delay comes from the fact that the BRAM blocks are only available at specific positions in the FPGA. Units which operate on memory cannot always be placed directly next to that BRAM blocks when connections to other BRAM blocks at different positions in the FPGA must also be created. This problem can either be resolved by changing the amount and position of the BRAM blocks on memory or adding more registers between the unit which wants to operate on the data and the BRAM. Additional registers would add more wait cycles to the state machines when the access to the memory cannot be pipelined. Also the position and amount of memory blocks cannot be changed by the user, but a simulation of the design for an ASIC could be used to evaluate whether it can be placed with a lower relative net delay.

6.2. Future Work

This work proposes concepts for a hardware unit for collective offload. We validated the applicability of our concepts by implementing them in a hardware description language. This gives us the ability to use standard tools to synthesize the hardware model for state of the art FPGAs. Doing so we get information about the maximum clock speed achievable by our design. Due to the complexity of such a hardware design we were not able to provide an FPGA prototype of our design which works together with the EXTOLL network. To use our design together with EXTOLL another module, called COMM unit in this thesis, is developed in another diploma thesis, which was not finished at the time of writing. Once all neccessary components are completed the next step is to carry out practical tests with the design and benchmark its performance. To enable such tests work has to be done to integrate a modified EXTOLL unit, which contains our design, into a standard host system. While we

developed software to generate binary schedules for interpretation by the GOAL interpreter developed in this thesis we did not develop a device driver for the proposed hardware because such a driver would also need the COMM unit for communication between the host and the GOAL interpreter.

Our design does not contain a arithmetic logic unit (ALU) which performs the local operations defined by GOAL. We did some preliminary analysis on the performance of arithmetic operations on FPGAs and concluded that such an ALU could provide a performance improvement for small reduction operations and propose a first idea on how such an ALU could be integrated into our design.

There are some points in the design space for a hardware collective offload unit that we did not explore in this thesis due to time or resource restrictions. For example we proposed a new point to point messaging protocol which eliminates the need to store information about unexpected messages at the cost of additional control messages compared with traditional rendezvous RDMA protocols. Another possibility to achieve the same goal would have been to retransmit unacknowledged messages at regular intervals. Those approaches are best compared in practice. Our design is modular, so replacing parts of it with different implementations it can be used as a research vehicle to gain more insight into the field of efficient collective offload. For the design proposed in this thesis we focused on the resources available on a Xilinx Virtex6 FPGA. When designing an application specific integrated circuit for collective offload, different resources might be available. In this case it could be viable to implement the matching unit based on ternary addressable content memory instead of iterating over memory.

6.3. Conclusions

To the best of our knowledge, this work is the first to describe an architecture for the complete offload of arbitrary collective operations. Most prior work in the field of collective offload is either proprietary and therefore the internal architecture of the hardware offload units are not published, does not allow the offload of arbitrary collective communication schemes or only offloads a part of the collective execution, such as message matching. Our work, combined with the EXTOLL network, is a platform suitable to extend the current knowledge about offloading collectives.

We showed that it is possible to interpret GOAL schedules in hardware and that the hardware units neccessary can be implemented on an FPGA in such a way that the whole design can be run at 212 MHz. Conservative estimates place standard cell ASICs

at five times the clock rate of FPGAs [HUR07], thus our design should achieve a 1 GHz clock frequency in a 90 nm standard cell ASIC. This would improve the performance of some parts of current message passing frameworks, such as message matching, by a factor of ten, compared to the currently available host-centric implementations.

We discovered that previously proposed solutions of how to execute large GOAL communication schedule with a limited amount of memory are not applicable in practice. However, this is not a shortcoming of the GOAL representation for collective functions, we suspect that all representations powerful enough to express arbitrary collectives have the same limitations when dealing with an execution window of limited size. To address this problem we propose a new point to point messaging protocol which eliminates the need to store information about unexpected messages. One of the features of GOAL is that it hides all details about how the collective will be executed from the user, the user just specifies a set of conditions for the execution. This however makes it harder for the user to reason about the absence of possible deadlocks in a GOAL schedule. We proposed algorithms that check a global GOAL schedule for possible deadlocks. This is much easier than checking an MPI program for deadlocks, as we do not need to analyze a programs control flow [Bro09] to get the data flow graph, we can directly analyze the GOAL graph.

Bibliography

[AHA+05] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow and Y. Zheng: *Optimization of MPI collective communication on BlueGene/L systems*, pp. 253–262, 2005.

[BGRU06] R. Brightwell, S. P. Goudy, A. Rodrigues and K. D. Underwood: *Implications of application usage characteristics for collective communication offload*, *International Journal of High Performance Computing and Networking*, vol. 4(3):pp. 104–116, 2006.

[BHK+02] J. Bruck, C. T. Ho, S. Kipnis, E. Upfal and D. Weathersby: *Efficient algorithms for all-to-all communications in multiport message-passing systems*, *Parallel and Distributed Systems, IEEE Transactions on*, vol. 8(11):pp. 1143–1156, 2002.

[BHSV+96] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo *et al.*: *VIS: A system for verification and synthesis*, in *Computer Aided Verification*, pp. 428–432, Springer, 1996.

[BPS+01] D. Buntinas, D. K. Panda, P. Sadayappan, Darius Buntinas and Dhabaleswar K. Panda: *Fast NIC-level Barrier over Myrinet/GM*, in *In Proceedings of the International Parallel and Distributed Processing Symposium*, 2001.

[BR96] S. Brown and J. Rose: *FPGA and CPLD architectures: A tutorial*, *Design & Test of Computers, IEEE*, vol. 13(2):pp. 42–57, 1996.

[Bro09] Greg Bronevetsky: *Communication-Sensitive Static Dataflow for Parallel Message Passing Applications*, in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pp. 1–12, IEEE Computer Society, Washington, DC, USA, 2009, ISBN 978-0-7695-3576-0.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella: *Nusmv 2: An opensource tool for symbolic model checking*, in *Computer Aided Verification*, pp. 241–268, Springer, 2002.

[CGH94] L. Clarke, I. Glendinning and R. Hempel: *The MPI message passing interface standard*, p. 213, 1994.

[CMH83] K. M. Chandy, J. Misra and L. M. Haas: *Distributed deadlock detection*, *ACM Transactions on Computer Systems (TOCS)*, vol. 1(2):p. 156, 1983, ISSN 0734-2071.

[Gol94] S. Golson: *State machine design techniques for Verilog and VHDL*, *Synopsys Journal of High-Level Design*, vol. 9:pp. 1–48, 1994.

[Gor04] S. Gorlatch: *Send-receive considered harmful: Myths and realities of message passing*, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 26(1):pp. 47–56, 2004, ISSN 0164-0925.

[GPS+10] R. L. Graham, S. Poole, P. Shamis, G. Bloch, N. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz and G. Shainer: *Overlapping computation and communication: Barrier algorithms and connectx-2 core-direct capabilities*, in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1–8, 2010.

[Gro02] W. Gropp: *MPICH2: A new start for MPI implementations*, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 37–42, 2002.

[GWS06] R. Graham, T. Woodall and J. Squyres: *Open MPI: A flexible high performance MPI*, *Parallel Processing and Applied Mathematics*, pp. 228–239, 2006.

[HGLR07a] T. Hoefler, P. Gottschling, A. Lumsdaine and W. Rehm: *Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations*, *Elsevier Journal of Parallel Computing (PARCO)*, vol. 33(9):pp. 624–633, Sep. 2007, ISSN 0167-8191.

[HGLR07b] T. Hoefler, P. Gottschling, A. Lumsdaine and W. Rehm: *Optimizing a conjugate gradient solver with non-blocking collective operations*, *Parallel Computing*, vol. 33(9):pp. 624–633, 2007.

[HK02] C. T. Ho and M. Y. Kao: *Optimal broadcast in all-port wormhole-routed hypercubes*, *Parallel and Distributed Systems, IEEE Transactions on*, vol. 6(2):pp. 200–204, 2002.

[HL08] T. Hoefler and A. Lumsdaine: *Message Progression in Parallel Computing - To Thread or not to Thread?*, in *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, IEEE Computer Society, Oct. 2008, ISBN 978-1-4244-2640, ISSN 1552-5244.

[HMLR07] T. Hoefler, T. Mehlan, A. Lumsdaine and W. Rehm: *Netgauge: A Network Performance Measurement Framework*, in *Proceedings of High Performance Computing and Communications, HPCC'07*, vol. 4782, pp. 659–671, Springer, Sep. 2007, ISBN 978-3-540-75443-5.

[HMMR06] T. Hoefler, T. Mehlan, F. Mietke and W. Rehm: *Adding Low-Cost Hardware Barrier Support to Small Commodity Clusters*, in *19th International Conference on Architecture and Computing Systems-ARCS*, vol. 6, pp. 343–350, 2006.

[Höf05] T. Höfler: *Evaluation of publicly available Barrier–Algorithms and Improvement of the Barrier–Operation for large–scale Cluster–Systems with special Attention on InfiniBandTMNetworks*, *URL: http://archiv. tuchemnitz. de/pub/2005/0073/data/diploma. pdf*, vol. 4, 2005.

[Hol97] G.J. Holzmann: *The model checker SPIN*, *Software Engineering, IEEE Transactions on*, vol. 23(5):pp. 279–295, 1997.

[HSL09a] T. Hoefler, T. Schneider and A. Lumsdaine: *The Effect of Network Noise on Large-Scale Collective Communications*, *Parallel Processing Letters (PPL)*, 2009.

[HSL09b] T. Hoefler, T. Schneider and A. Lumsdaine: *The impact of network noise at large-scale communication performance*, in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–8, IEEE, 2009.

[HSL09c] T. Hoefler, C. Siebert and A. Lumsdaine: *Group Operation Assembly Language - A Flexible Way to Express Collective Communication*, in *ICPP-2009 - The 38th International Conference on Parallel Processing*, IEEE, Sep. 2009, ISBN 978-0-7695-3802-0.

[HSL10] T. Hoefler, T. Schneider and A. Lumsdaine: *Characterizing the influence of system noise on large-scale applications by simulation*, in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, IEEE Computer Society, 2010.

[HUR07] K. S. Hemmert, K. D. Underwood and A. Rodrigues: *An architecture to perform NIC based MPI matching*, in *Cluster Computing, 2007 IEEE International Conference on*, pp. 211–221, 2007.

[HYK03] S. Habata, M. Yokokawa and S. Kitawaki: *The earth simulator system*, *NEC Research and Development*, vol. 44(1):pp. 21–26, 2003.

[Hym66] Harris Hyman: *Comments on a problem in concurrent programming control*, *Commun. ACM*, vol. 9:pp. 45–, January 1966, ISSN 0001-0782. URL `http://doi.acm.org/10.1145/365153.365167`

[HZ07] T. Hoefler and G. Zerah: *Transforming the high-performance 3d-FFT in ABINIT to enable the use of non-blocking collective operations*, Tech. rep., Feb. 2007.

[Inc09] Xilinx Inc.: *XST User Guide v11.3 UG627*, pp. 189–191, September 2009.

[Inc11a] Xilinx Inc.: *Floating-Point Operator v6.0 DS816*, June 2011.

[Inc11b] Xilinx Inc.: *LogiCORE IP Block Memory Generator v6.1 DS512*, March 2011.

[Inc11c] Xilinx Inc.: *LogiCORE IP Divider Generator v4.0 DS819*, June 2011.

[Inc11d] Xilinx Inc.: *MicroBlaze Processor Reference Guide v13.3 UG081*, October 2011.

[Inc11e] Xilinx Inc.: *Virtex-6 Family Overview v2.3 DS150*, March 2011.

[Inc11f] Xilinx Inc.: *Virtex-6 FPGA Memory Resources v1.6 DS819*, April 2011.

[LWP04] J. Liu, J. Wu and D.K. Panda: *High performance RDMA-based MPI implementation over InfiniBand*, *International Journal of Parallel Programming*, vol. 32(3):pp. 167–198, 2004.

[MMHR07] F. Mietke, T. Mehlan, T. Hoefler and W. Rehm: *Design and Evaluation of a 2048 Core Cluster System*, *Kommunikation in Clusterrechnern und Clusterverbund-systemen*, p. 40, 2007.

[NFG+] M. Nüssle, H. Fröning, A. Giese, H. Litz, D. Slogsnat and U. Brüning: *A Hypertransport based low-latency reconfigurable testbed for message-passing developments*, in *Proceedings of the 2nd Workshop Kommunikation in Clusterrechnern und Clusterverbundsystemen (KiCC 2007)*, pp. 1–6.

[NGFB09] M. Nüssle, B. Geib, H. Fröning and U. Brüning: *An FPGA-based custom high performance interconnection network*, in *2009 International Conference on Reconfigurable Computing and FPGAs*, pp. 113–118, IEEE, 2009.

[NI10] A. Nomura and Y. Ishikawa: *Design of Kernel-Level Asynchronous Collective Communication*, *Recent Advances in the Message Passing Interface*, pp. 92–101, 2010.

[NSB09] M. Nussle, M. Scherer and U. Bruning: *A resource optimized remote-memory-access architecture for low-latency communication*, in *Parallel Processing, 2009. ICPP'09. International Conference on*, pp. 220–227, 2009.

[Pak08] S. Pakin: *Receiver-initiated message passing over RDMA networks*, in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–12, IEEE, 2008.

[Pfi01] G.F. Pfister: *An introduction to the InfiniBand architecture*, *High Performance Mass Storage and Parallel I/O*, pp. 617–632, 2001.

[PKP03] F. Petrini, D. J. Kerbyson and S. Pakin: *The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q*, in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, p. 55, 2003.

[Rie08] Riesen, R.E. and Pedretti, K.T. and Brightwell, R. and Barrett, B.W. and Underwood, K.D. and Hudson, T.B. and Maccabe, A.B.: *The Portals 4.0 message passing interface*, Sandia National Laboratories, April 2008, technical Report SAND2008-2639.

[SBM+05] S. Sur, U.K.R. Bondhugula, A. Mamidala, H.W. Jin and D.K. Panda: *High performance RDMA based all-to-all broadcast for InfiniBand clusters*, *High Performance Computing–HiPC 2005*, pp. 148–157, 2005.

[SEHR11] Timo Schneider, Sven Eckelmann, Torsten Hoefler and Wolfgang Rehm: *Kernel-Based Offload of Collective Operations - Implementation, Evaluation and Lessons Learned*, in *Euro-Par (2)* (edited by Emmanuel Jeannot, Raymond Namyst and Jean Roman), vol. 6853 of *Lecture Notes in Computer Science*, pp. 264–275, Springer, 2011, ISBN 978-3-642-23396-8.

[SGB07] D. Slogsnat, A. Giese and U. Brüning: *A versatile, low latency HyperTransport core*, in *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pp. 45–52, ACM, 2007.

[shi06] *High Performance RDMA Protocols in HPC*, in *Proceedings, 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, Springer-Verlag, Bonn, Germany, September 2006.

[SJCP06] S. Sur, H.W. Jin, L. Chai and D.K. Panda: *RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits*, in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 32–39, ACM, 2006.

[SKSP10] H. Subramoni, K. Kandalla, S. Sur and D. K. Panda: *Design and Evaluation of Generalized Collective Communication Primitives with Overlap using ConnectX-2 Offload Engine*, in *2010 18th IEEE Symposium on High Performance Interconnects*, pp. 40–49, 2010.

[SSP97] R. Sivaram, C. B. Stunkel and D. K. Panda: *A reliable hardware barrier synchronization scheme*, in *ipps*, p. 274, 1997.

[SWP01] P. Shivam, P. Wyckoff and D. Panda: *EMP: zero-copy OS-bypass NIC-driven gigabit ethernet message passing*, p. 57, 2001.

[UHR+05] K.D. Underwood, K.S. Hemmert, A. Rodrigues, R. Murphy and R. Brightwell: *A hardware acceleration unit for mpi queue processing*, in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pp. 96b–96b, IEEE, 2005.

[VFD00] S. S. Vadhiyar, G. E. Fagg and J. Dongarra: *Automatically tuned collective communications*, p. 3, 2000.

[VGL+11] M. G. Venkata, R. L. Graham, J. S. Ladd, P. Shamis, I. Rabinovitz, V. Filipov and G. Shainer: *ConnectX-2 CORE-Direct Enabled Asynchronous Broadcast Collective Communications*, in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 781–787, 2011.

[Vol10] Toni Volkmer: *Development of Concepts and Implementation Approaches for the Utilization of Processors in Host-Coupled FPGAs*, Master's thesis, TU Chemnitz, 2010.

[YBP03] Weikuan Yu, Darius Buntinas and Dhabaleswar K. Panda: *High Performance and Reliable NIC-Based Multicast over Myrinet/GM-2*, in *IN INTL CONFERENCE ON PARALLEL PROCESSING, ICPP 2003*, 2003.

A. Send Protocol State Machine

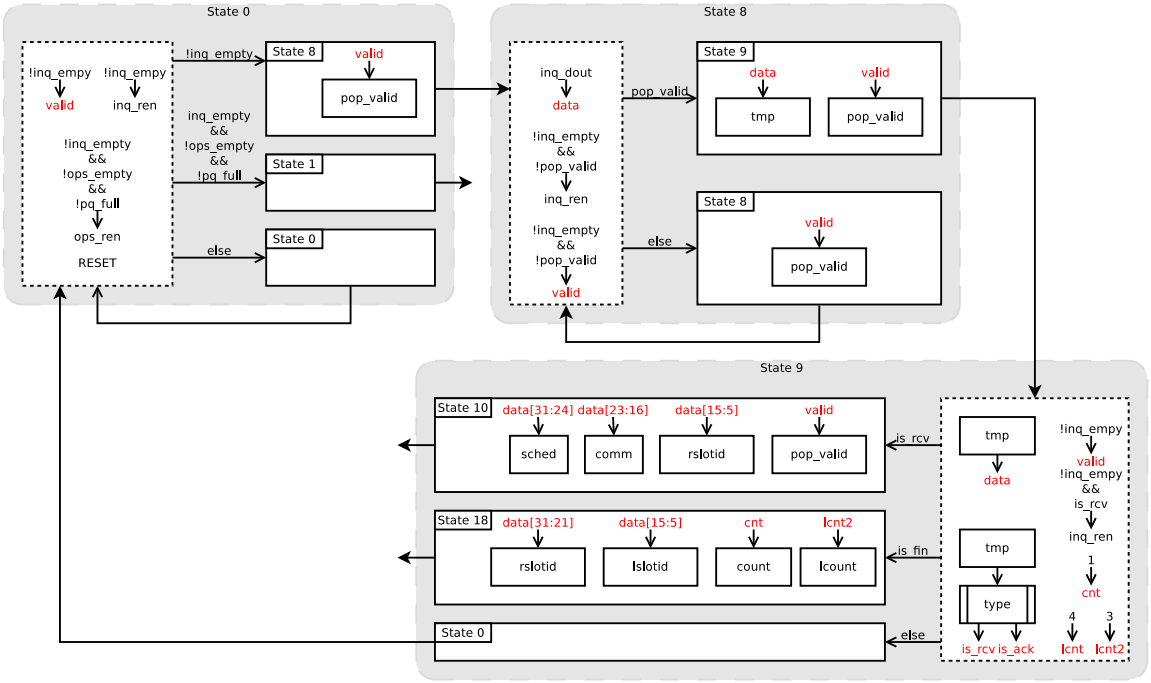


Figure A.1.: Finite State Machine of the Send Protocol (detection of packet)

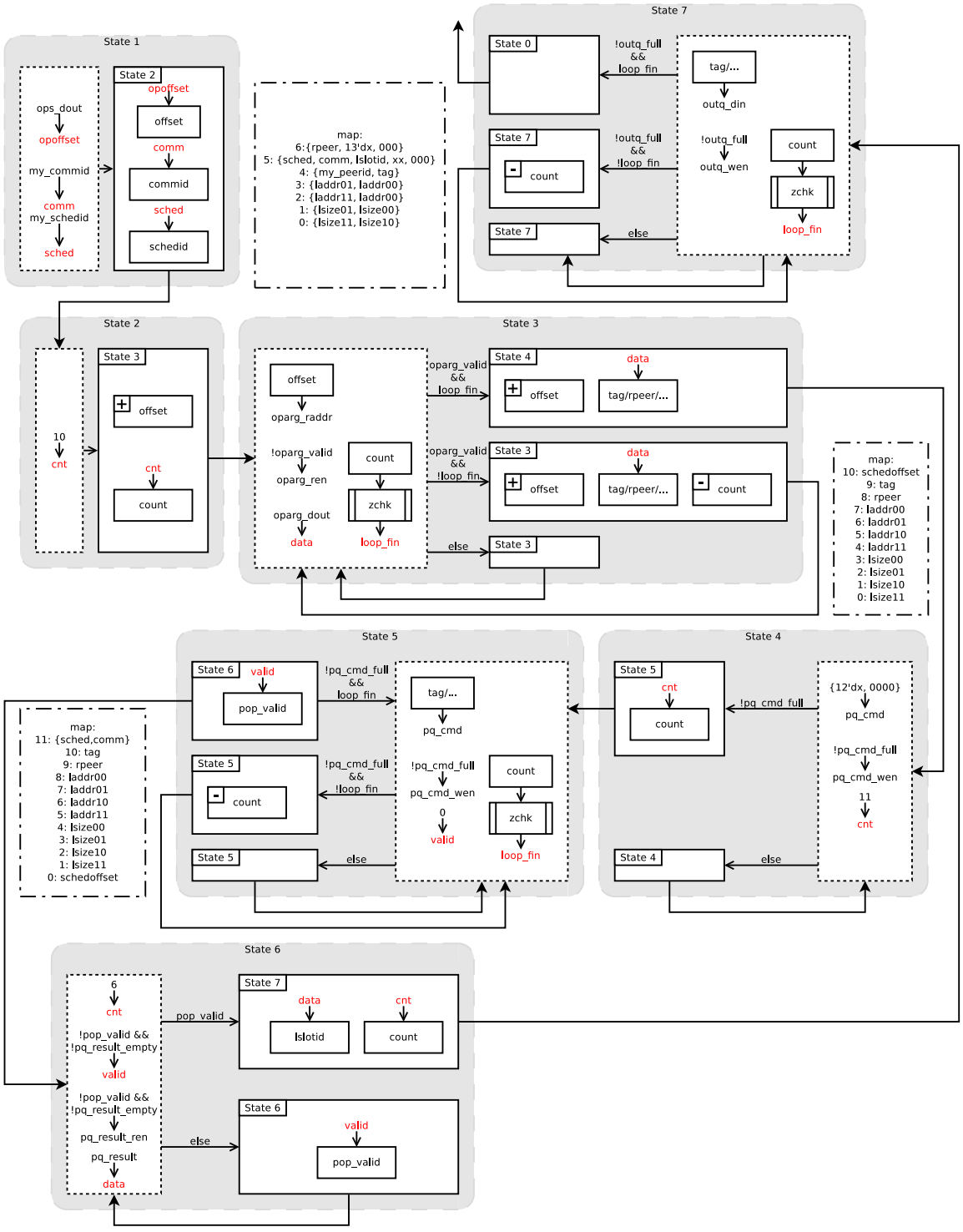


Figure A.2.: Finite State Machine of the Send Protocol (slot initialization and send of SND_RDY)

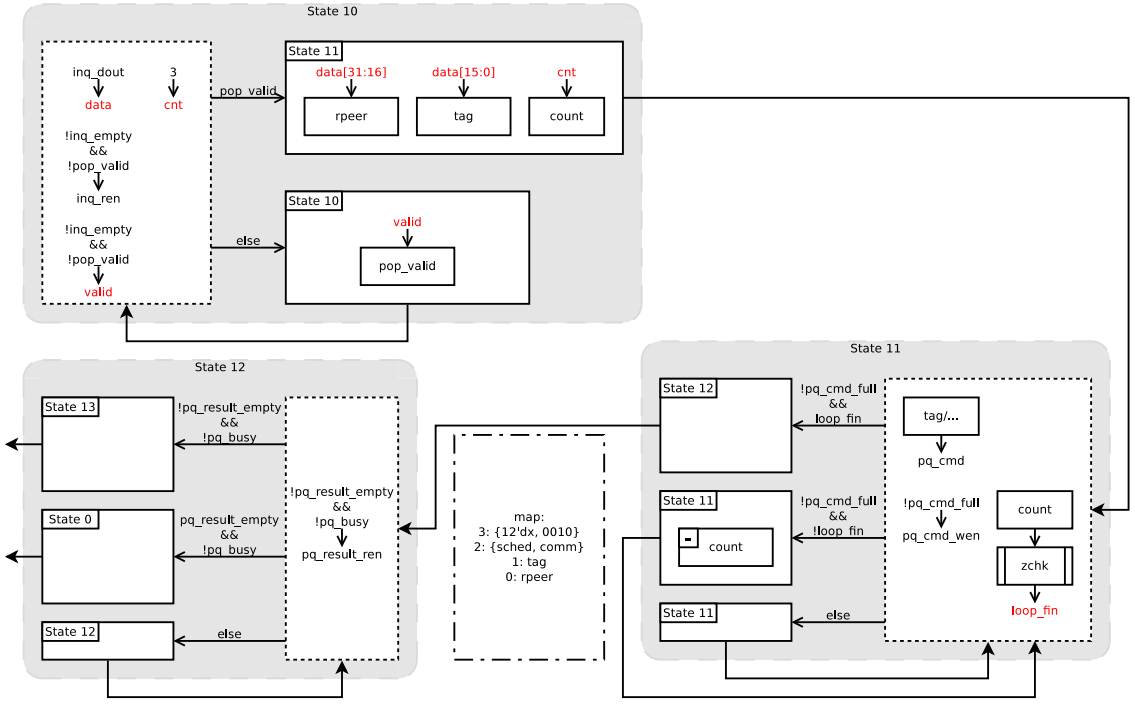


Figure A.3.: Finite State Machine of the Send Protocol (matching of received RCV_RDY)

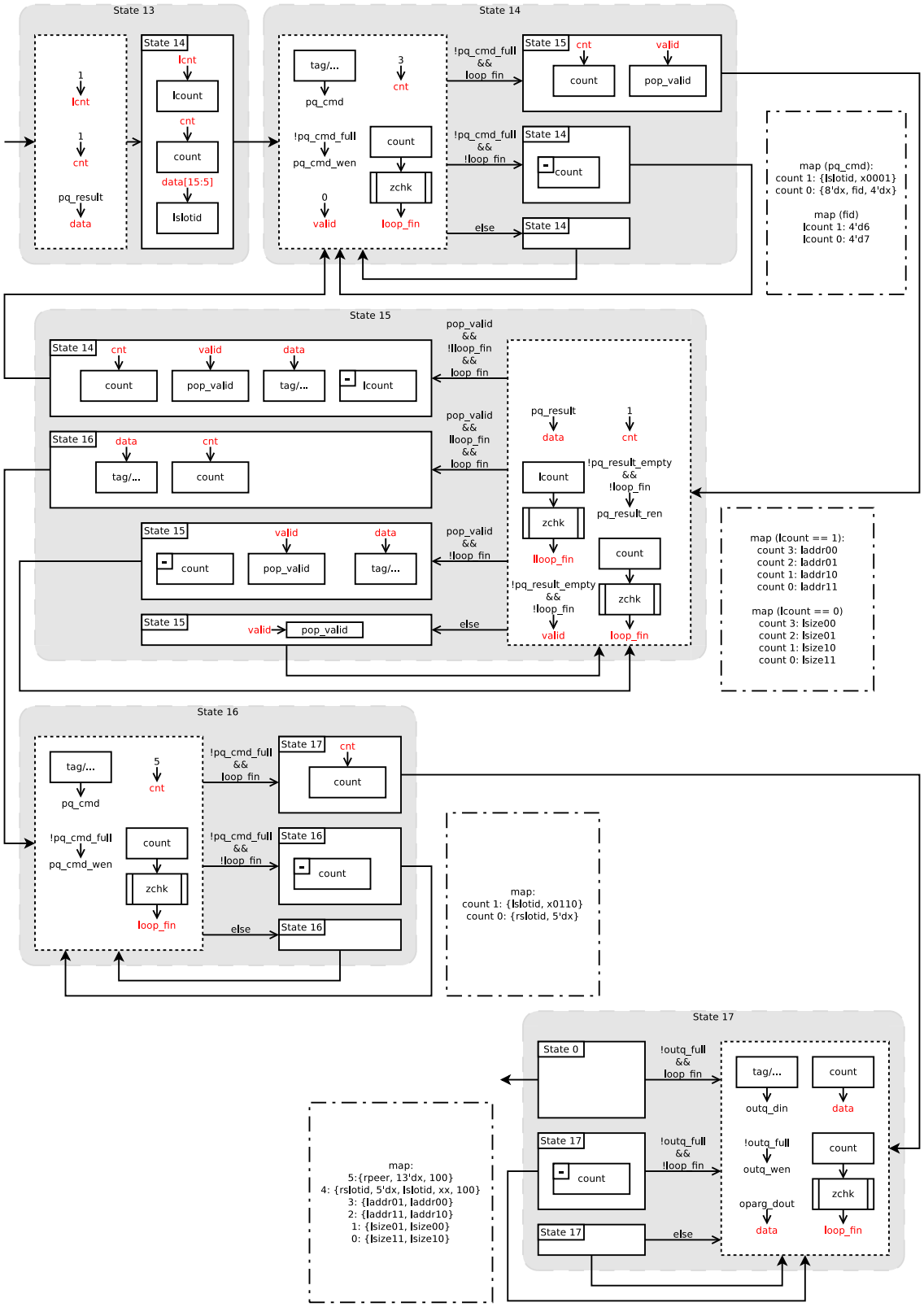


Figure A.4.: Finite State Machine of the Send Protocol (ACK reply after RCV_RDY)

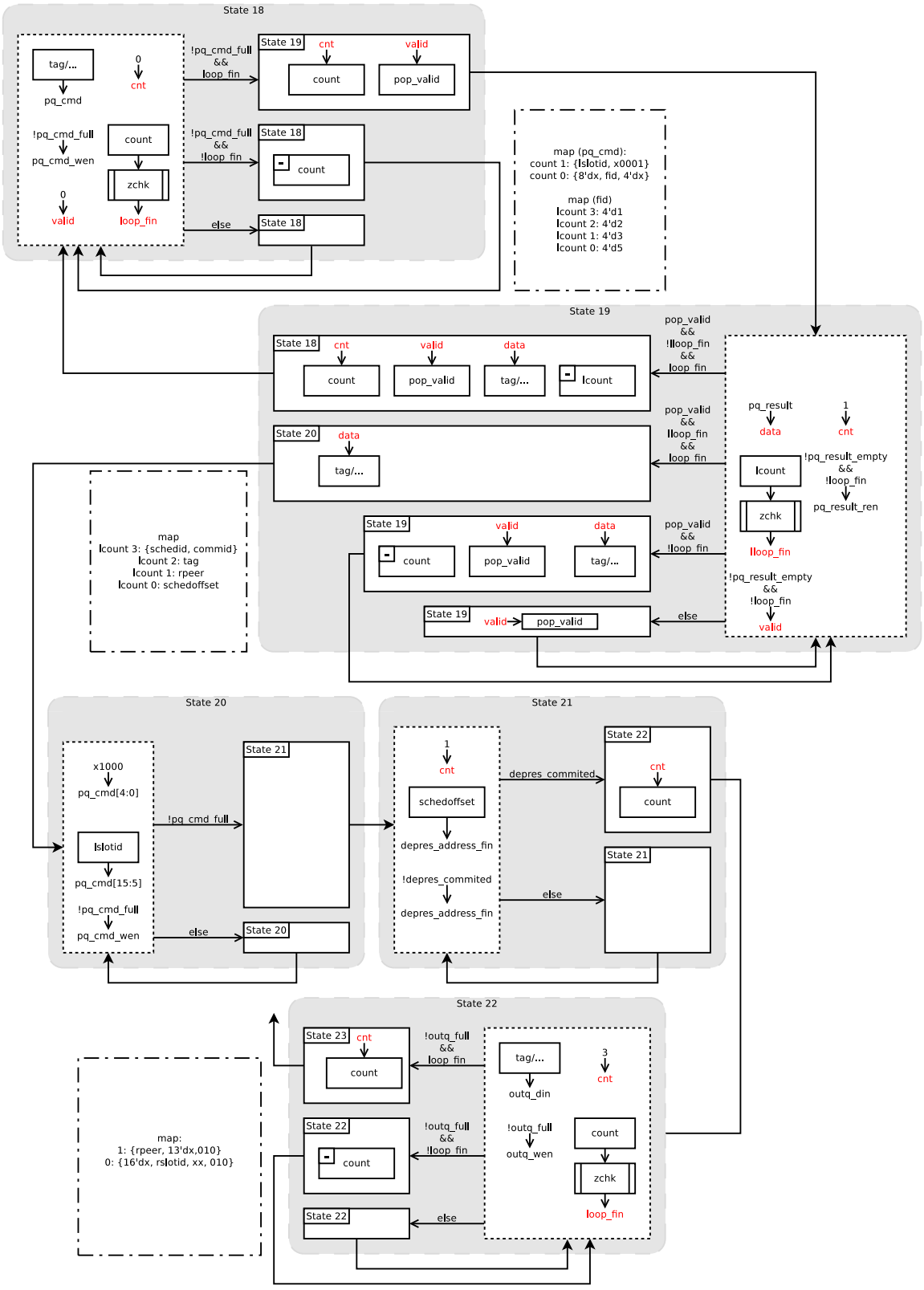


Figure A.5.: Finite State Machine of the Send Protocol (FIN processing)

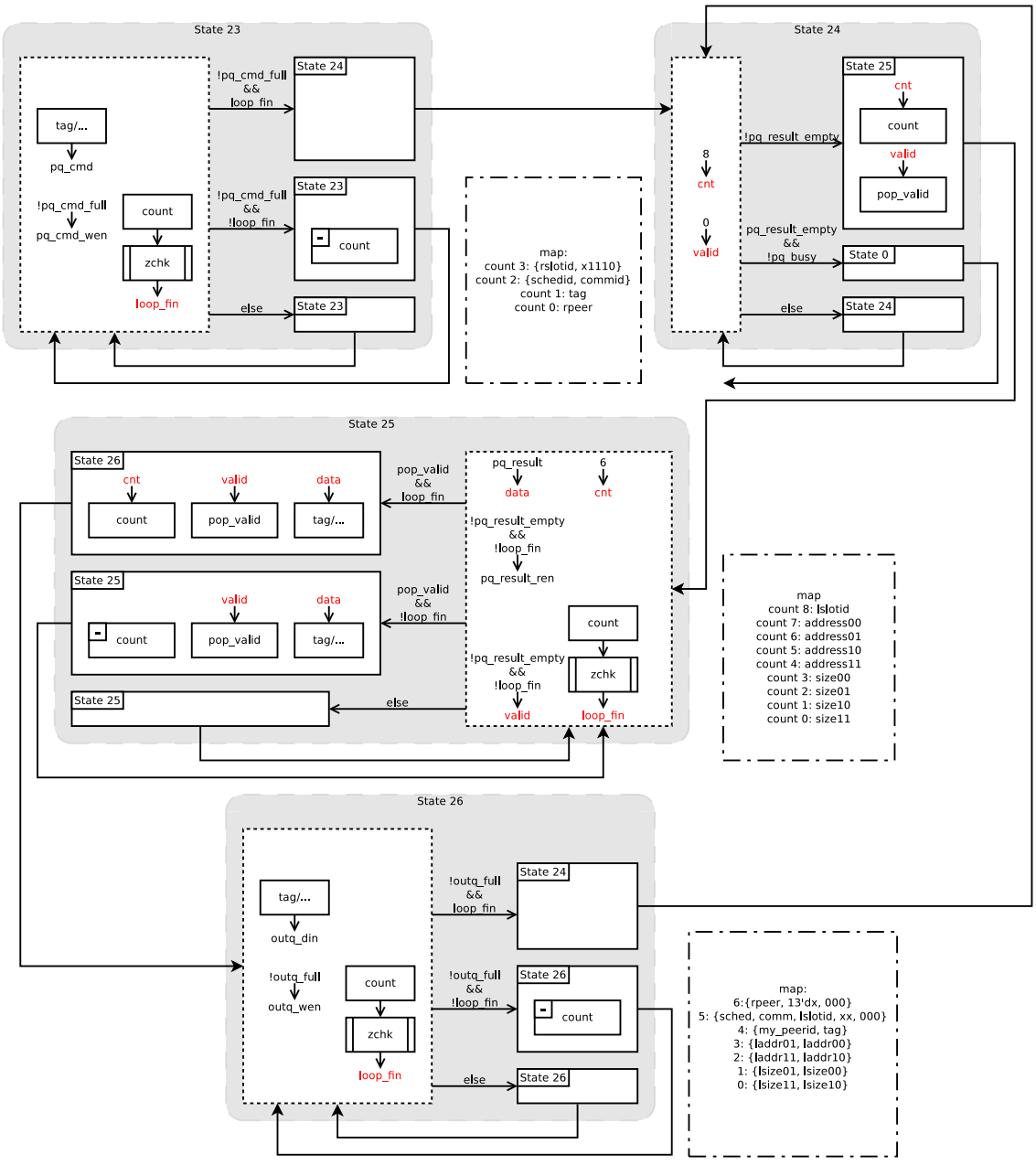


Figure A.6.: Finite State Machine of the Send Protocol (resent of possibly dropped SND_RDY)

B. Receive Protocol State Machine

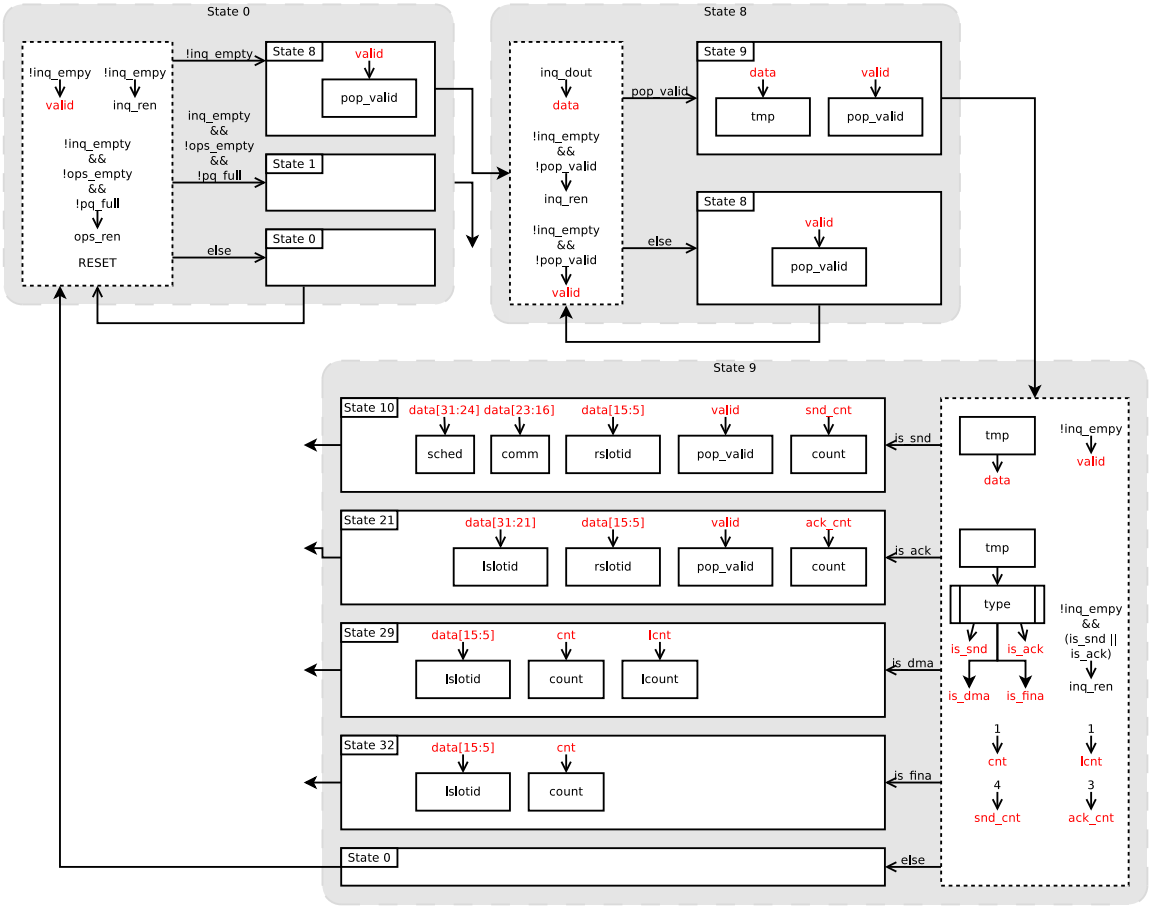


Figure B.1.: Finite State Machine of the Receive Protocol (detection of packet)

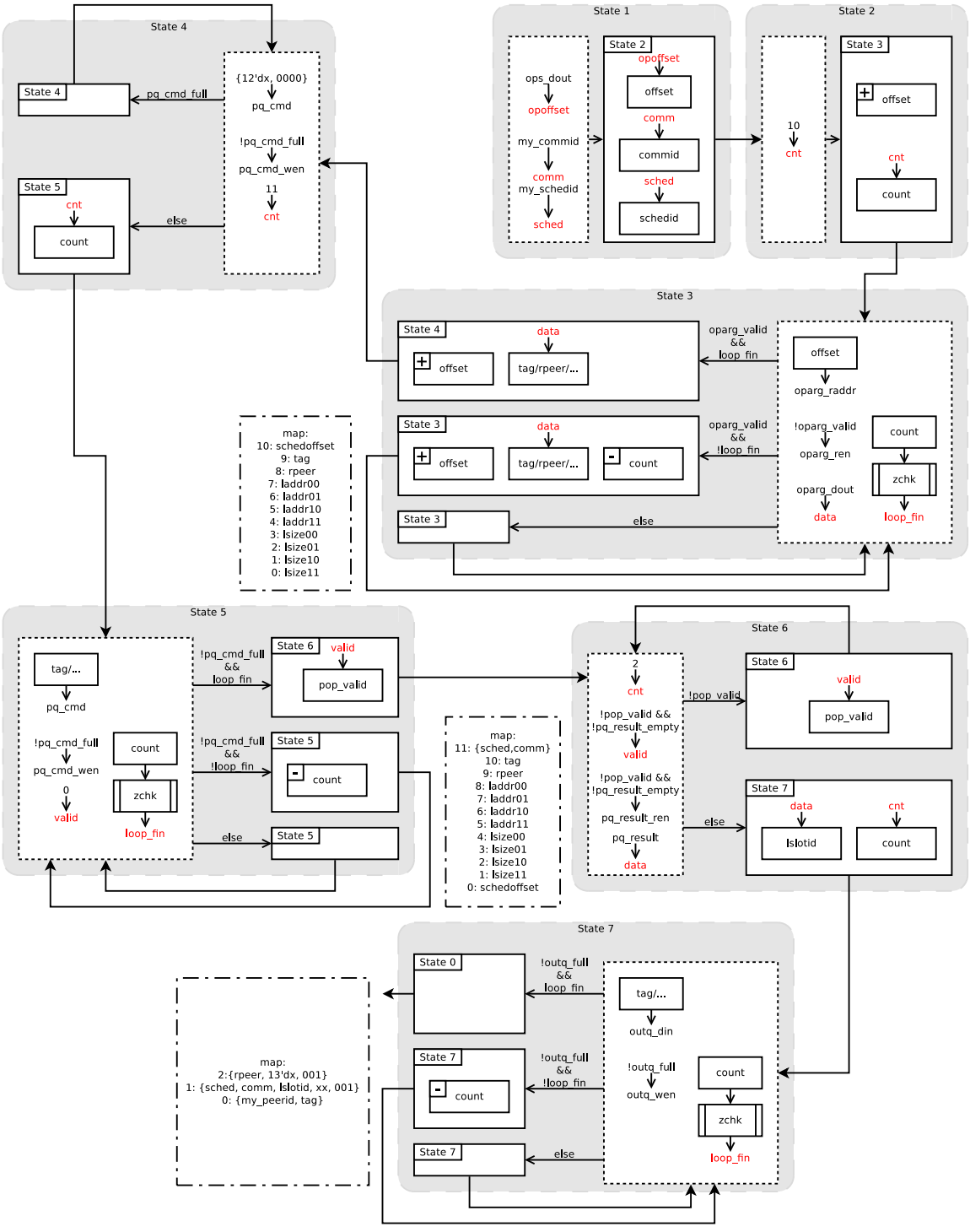


Figure B.2.: Finite State Machine of the Receive Protocol (slot initialization and send of RCV_RDY)

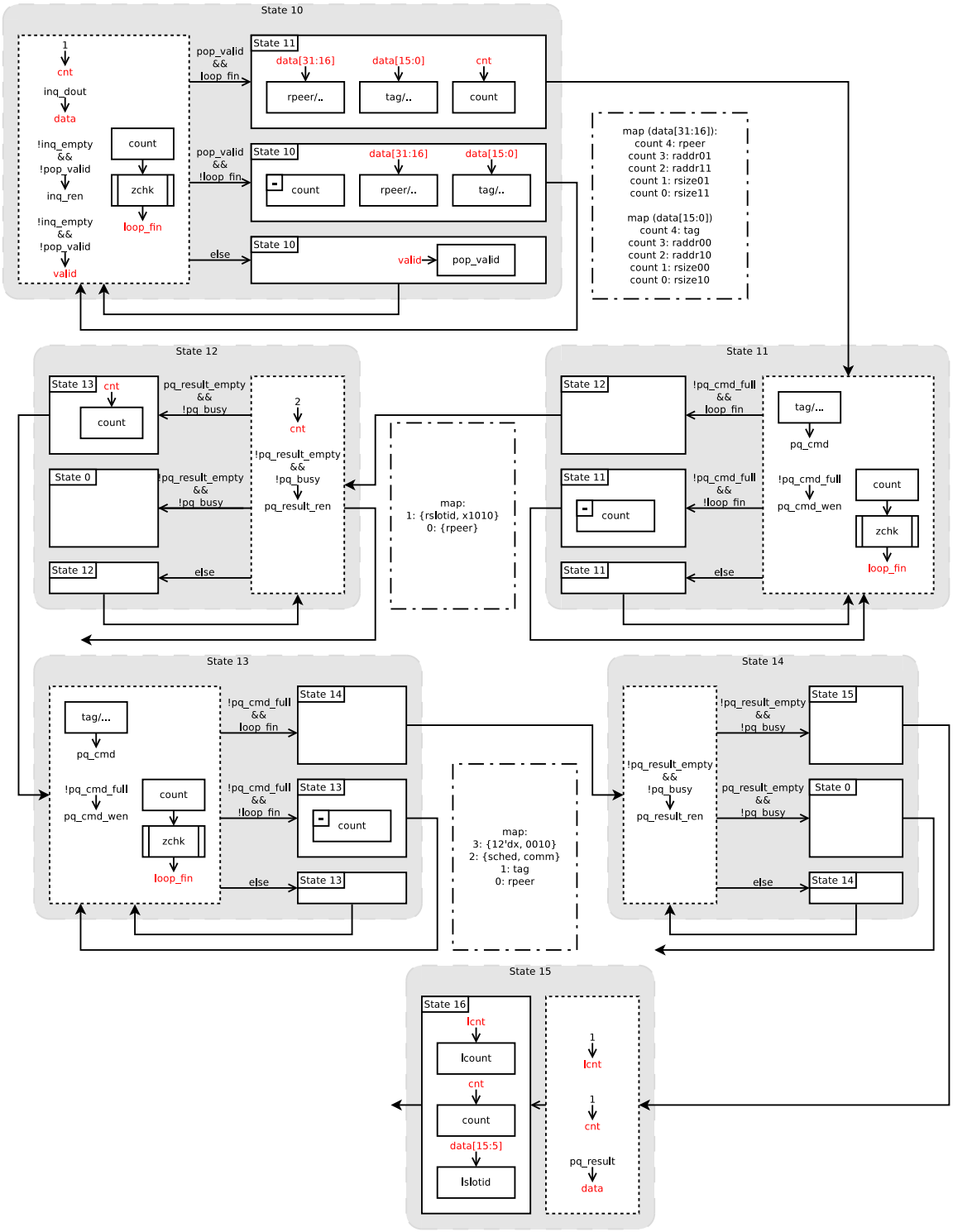


Figure B.3.: Finite State Machine of the Receive Protocol (matching of received SND_RDY)

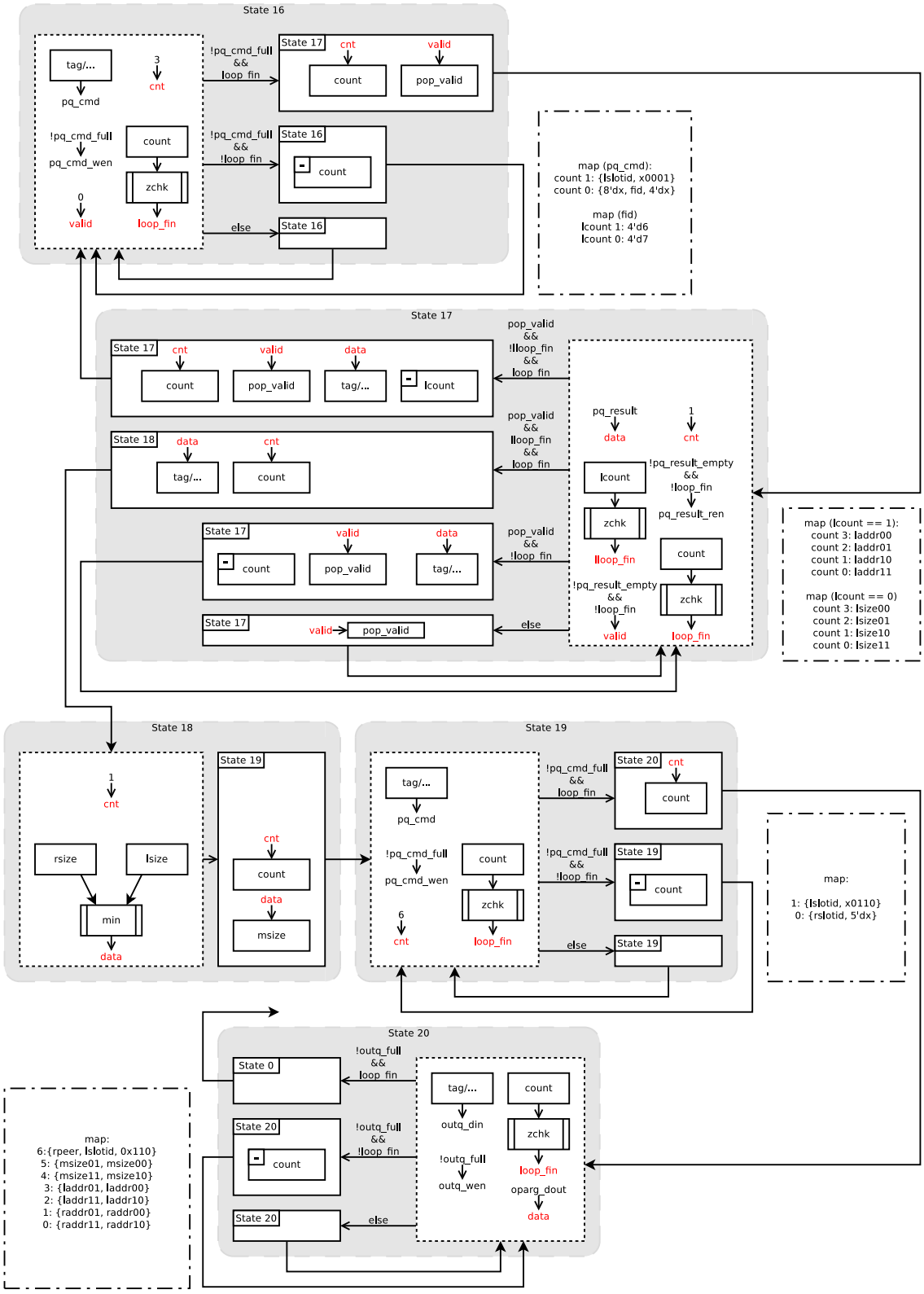


Figure B.4.: Finite State Machine of the Receive Protocol (DMA_GET reply after SND_RDY)

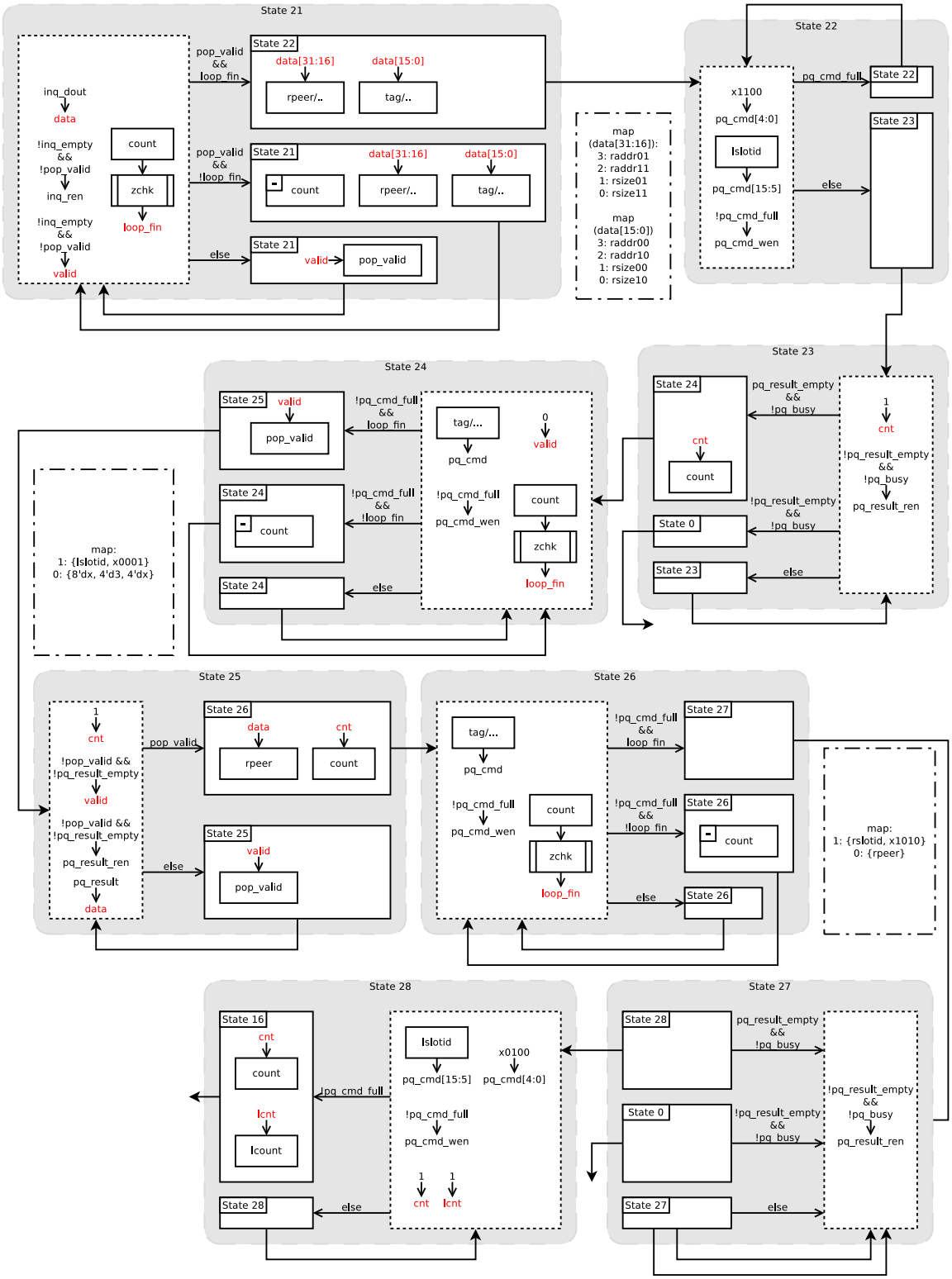


Figure B.5.: Finite State Machine of the Receive Protocol (matching of received ACK)

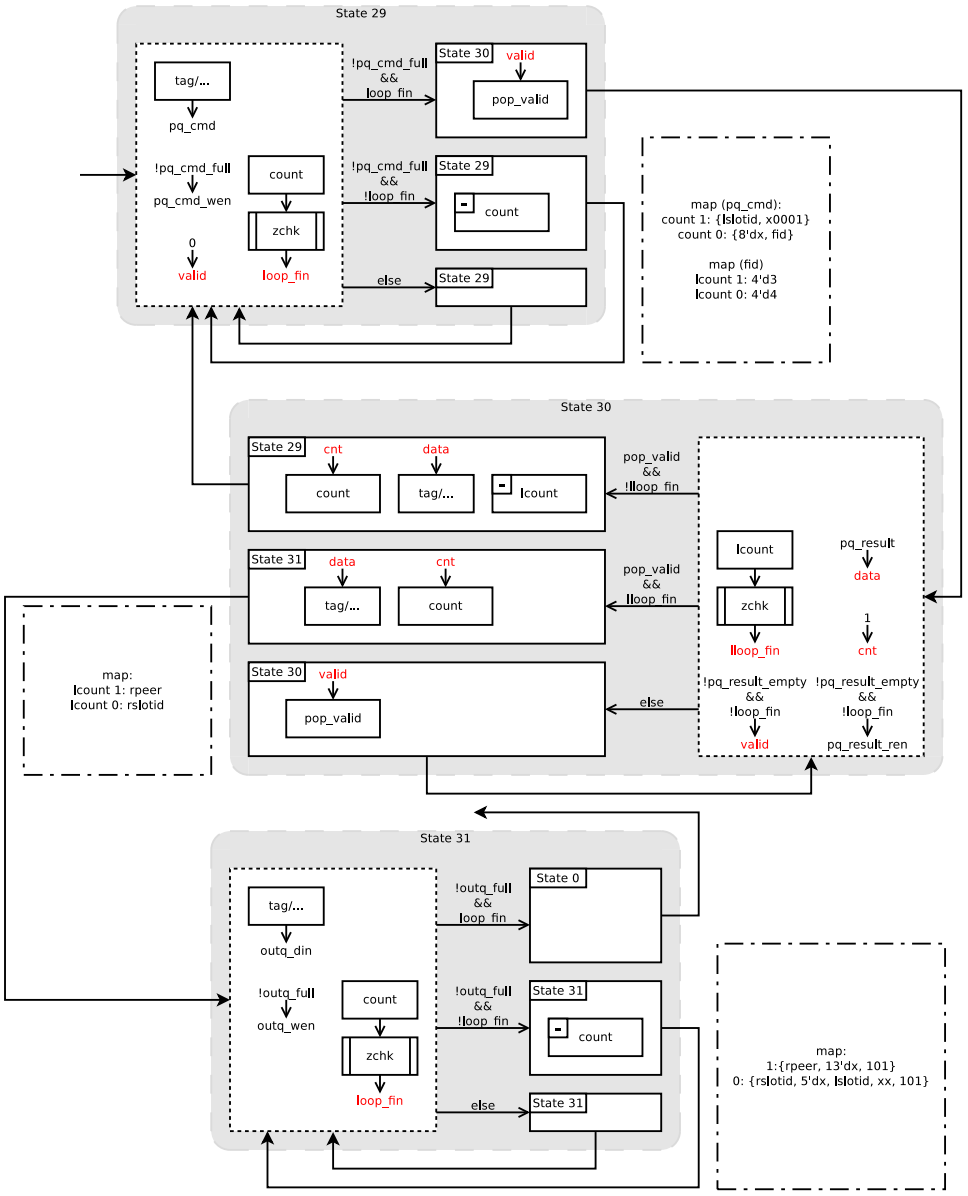


Figure B.6.: Finite State Machine of the Receive Protocol (processing of a DMA_FIN packet)

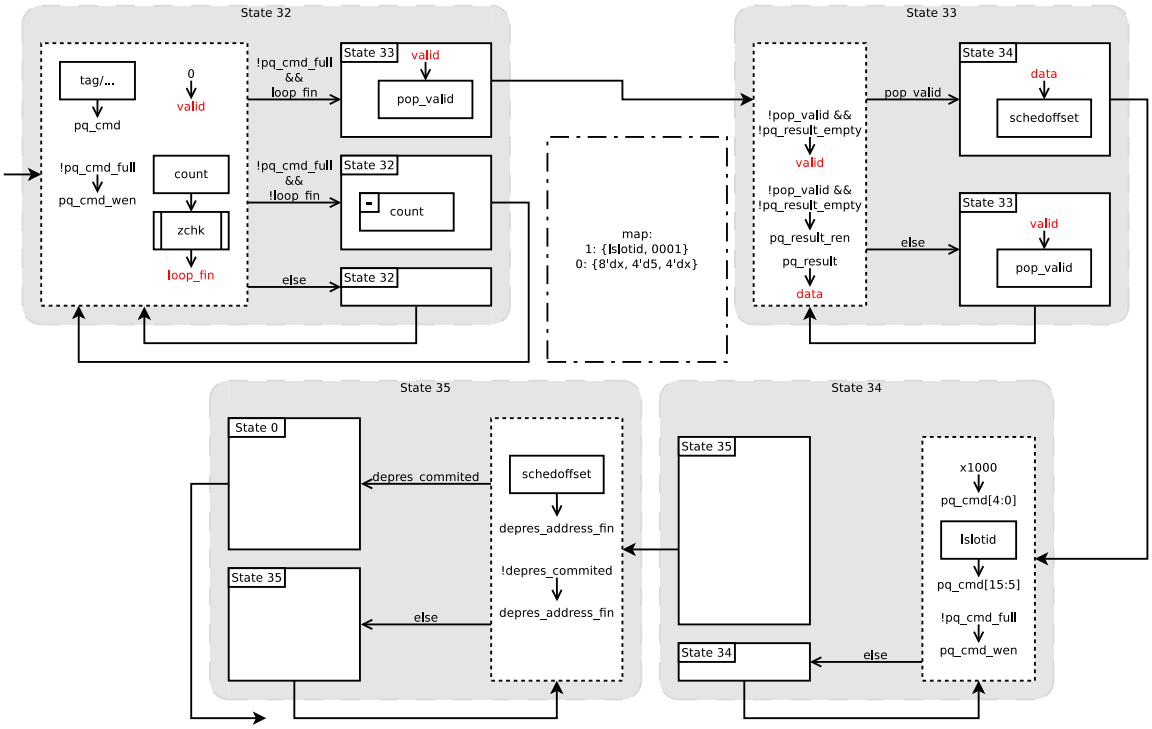


Figure B.7.: Finite State Machine of the Receive Protocol (FIN_ACK processing)

C. CD-ROM Content

The CD-ROM attached to this thesis includes the content listed in Table C.1.

Directory	Description
thesis/	LaTeX source and PDF of this document
src/spin/	SPIN models of the point to point protocol (c.f. Section 3.6.4)
src/ompi_queuebench/	Benchmarks described in Section 4.1
src/scripts/	State-diagram-to-verilog translator, Script to generate Verilog memory initialization code from binary data
src/verilog/schedule_interpreter/	Verilog code of the schedule interpreter unit
src/verilog/localop_test/	Verilog code for the local operation performance evaluation
src/sched_gen/	GOAL schedule generators for the collectives described in Section 3.3
src/simulation/	A cycle-accurate C++ simulation of the dependency resolver
src/statemachines/	The statemachines of several functional units used in the schedule interpreter design
comm_unit_spec/	Specification of the COMM unit
talks/fog-2011-*/	Talks we gave at the internal research group meetings

Table C.1.: CD-ROM Table of Content

D. Task Separation

1. Task Description	both
2. Introduction	Timo Schneider
2.4. Group Operation Assembly Language	both
3. Dealing with Constrained Resources	both
3.1. Hardware Limitations	Sven Eckelmann
3.2. Common Collective Functions in GOAL	Sven Eckelmann
3.3. Schedule Representation for the Hardware GOAL Interpreter	Sven Eckelmann
3.4. Executing Large Schedules using a small amount of Memory	both
3.4.1. Limits of Previously Suggested Approaches	both
3.4.2. Testing for Deadlocks in Schedules	both
3.4.3. Transforming Process Local Schedules into Global Schedules	Timo Schneider
3.4.4. Predetermined Buffer Locations	both
3.5. Queueing Active Operations in Hardware	both
3.6. Designing a Low-Memory-Footprint Point to Point Protocol	both
3.6.1. Arrival Times	Timo Schneider
3.6.2. Eager Protocol	Timo Schneider
3.6.3. Rendezvous Protocol	Timo Schneider
3.6.4. A Protocol without an Unexpected Queue	both
3.7. Protocol Verification	both
3.7.1. Capabilities of the Model Checker SPIN	Timo Schneider
3.7.2. Modeling the Protocol	both
3.7.3. Limitations of the Basic Protocol	Sven Eckelmann
4. The Matching Problem	Timo Schneider
5. The GOAL Interpreter	Sven Eckelmann
6. Evaluation	Timo Schneider
6.1. Performance Analysis	Sven Eckelmann
6.2. Future Work	Timo Schneider
6.3. Conclusions	Timo Schneider

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Chemnitz, den December 1, 2011

Timo Schneider

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Chemnitz, den December 1, 2011

Sven Eckelmann